Microsoft | Architecture

Guía de Arquitectura N-Capas orientada al Dominio con .NET 4.0

César de la Torre Llorente (Microsoft)
Unai Zorrilla Castro (Plain Concepts)
Javier Calvarro Nelson (Microsoft)
Miguel Ángel Ramos Barroso (Microsoft)

1ª EDICION

Microsoft Contenidos para Arquitectos

Krasis PRESS

Microsoft Ibérica ha detectado en diversos clientes la necesidad de disponer de una "Guía de Arquitectura base .NET" que sirva para marcar unas líneas maestras de diseño e implementación a la hora de desarrollar aplicaciones .NET complejas. Este marco de trabajo común (en muchas empresas denominado "Libro Blanco") define un camino para diseñar e implementar aplicaciones empresariales con un volumen importante de lógica de negocio. Seguir estas guías ofrece importantes beneficios en cuanto a calidad, estabilidad y especialmente un incremento en la facilidad del mantenimiento futuro de las aplicaciones, debido al desacoplamiento entre sus componentes, así como por la homogeneidad y similitudes de los diferentes desarrollos.

Microsoft Ibérica define el presente 'Libro de Arquitectura Marco' como patrón y modelo base, sin embargo, en ningún caso este marco debe ser inalterable. Al contrario, se trata del primer peldaño de una escalera, un acelerador inicial, que debería ser personalizado y modificado por cada organización que lo adopte, enfocándolo hacia necesidades concretas, adaptándolo y agregándole funcionalidad específica según el mercado objetivo, etc.

Audiencia
Este documento está dirigido a las personas involucradas en todo el ciclo de vida de las aplicaciones a medida desarrolladas. Especialmente los siguientes perfiles:

- Arquitecto de Software
- Desarrollador

Objetivos de la Arquitectura marco .NET
Esta guía pretende describir una arquitectura marco sobre la que desarrollar las aplicaciones a medida y establece un conjunto de normas, mejores prácticas y guías de desarrollo para utilizar .NET de forma adecuada y homogénea.

Autores
César de la Torre Llorente -Microsoft
Unai Zorrilla Castro (MVP) - Plain Concepts
Javier Calvarro Nelson - Microsoft
Miguel Ángel Ramos Barroso - Microsoft

Autores parciales
Cristian Manteiga Pacios - Plain Concepts
Fernando Cortés Hierro - Plain Concepts
Israel García Mesa - Microsoft

Colaboradores
Pierre Milet LLobet - Microsoft
Ricardo Minguez Pablos (Rido) - Microsoft
Hadi Hariri (MVP) - JetBrains
Roberto Gonzalez (MVP) - Renacimiento
Juan Cid - Avanade
Lalo Steinmann - Microsoft

ISBN 978-84-936696-3-8
9 788493 669638 >

Microsoft | Architecture
Krasis Press


¿Te interesa este libro?
Hazte con él en papel en http://shop.campusmvp.com
campus MVP
feed your brain®
Formación online
Preparación de certificaciones
Libros especializados
Servicios para empresas
en tecnologías Microsoft
krasis PRESS
Los libros que lo saben todo sobre tecnologías Microsoft
infórmate ya
902 876 475 www.campusmvp.com
http://www.krasis.com
krasis
Microsoft Partner

Guía de Arquitectura N-Capas Orientada al Dominio con .NET 4.0

Autores

César de la Torre Llorente (Microsoft)
Unai Zorrilla Castro (Plain Concepts)
Javier Calvarro Nelson (Microsoft)
Miguel Ángel Ramos Barroso (Microsoft)

Autores parciales

Cristian Manteiga Pacios (Plain Concepts)
Fernando Cortés Hierro (Plain Concepts)
Israel García Mesa (Microsoft)

Colaboradores

Pierre Milet LLobet (Microsoft)
Ricardo Minguez Pablos (Rido) (Microsoft)
Hadi Hariri (MVP) (JetBrains)
Roberto Gonzalez (MVP) (Renacimiento)
Juan Cid (Avanade)
Lalo Steinmann (Microsoft)

GUÍA DE ARQUITECTURA N-CAPAS ORIENTADA AL DOMINIO CON .NET 4.0

EDITADO por Krasis Consulting, S. L. **www.krasis.com**

ISBN: 978-84-936696-3-8
Depósito Legal: M-13152-2010

Impreso en España-Printed in Spain

Agradecimientos

César de la Torre

‘*Dedico este libro especialmente a mi familia, que ha sufrido el trabajo de innumerables fines de semana trabajando en ello. También lo dedico a nuestra compañía Microsoft y específicamente a Microsoft Ibérica, porque con este trabajo hemos aunado esfuerzos de diferentes áreas muy complementarias. “One-Microsoft!”.*

Lo siguiente son los comentarios de mi familia sobre este libro… ;-)

Mi mujer, Marta:
A ver si lo acabas que tenemos pendiente muchas cosas de la casa o irnos de escapadas más a menudo…

Mi hija Erika (9 años):
Papi, trabajas mucho y esto no se entiende nada…

Mi hijo Adrián (6 años):
No sé.., ¿jugamos a la XBOX?
’

Unai Zorrilla

‘A Lucia y Maria, mi familia, por su ‘inmerecida’ paciencia con mis maratonianas jornadas y mis continuos viajes…’

Javier Calvarro

‘A mi abuela Teresa. Te dedico todo el esfuerzo y dedicación que he puesto en estas páginas.’

Miguel Ángel Ramos Barroso

“Para Rosario; mi compañera, mi amiga, mi amor, mi aliento, mi vida. Sólo quince años juntos, y aún nos queda mucho por compartir.”

Contenido

CAPA DE SERVICIOS DISTRIBUIDOS.. 259

CAPA DE PRESENTACIÓN 345

CAPAS DE INFRAESTRUCTURA TRANSVERSAL 395

Prólogos

Prólogo de Enrique Fernandez-Laguilhoat
(*Director División de Plataforma y Desarrollo en Microsoft Ibérica*)

No es por casualidad que el sector de la informática ha imitado al de la construcción utilizando las apelaciones de Arquitecto y de Arquitectura. Al igual que en las grandes obras de construcción, para garantizar el éxito en el desarrollo de un aplicativo software se requiere antes que nada de una buena definición de la estructura que se va a seguir, de los distintos elementos o módulos que se van a construir y de cómo interactúan entre ellos de forma segura y eficaz. Un mal trabajo de arquitectura lleva en muchos casos al fracaso del proyecto, y al contrario, si el arquitecto de software hace bien su cometido, el producto resultante tenderá a ser robusto, el tiempo y esfuerzo para desarrollarlo más bajo, y algo muy importante, la facilidad para ampliar o extender el desarrollo en un futuro será mucho más alta.

Esta guía viene a cubrir un área muy importante en el mundo del desarrollo. De la mano de un grupo notable de profesionales de software y liderados por César de la Torre, uno de los principales Arquitectos de Software con los que cuenta Microsoft, se ofrece una visión exhaustiva y sistemática de cómo deber abordarse un desarrollo en capas utilizando la tecnología .Net. Y además, lo hace en perfecto Castellano viniendo a saldar una vieja deuda que Microsoft Ibérica tenía con los desarrolladores de habla hispana. Si desarrollar con el "framework" .Net siempre ha sido fácil y altamente productivo, la llegada de esta guía ofrece además una ayuda altamente estructurada que facilita la definición de la arquitectura y el modelado de la aplicación.

Ha sido un placer ver durante varios meses la ilusión (y las largas horas de trabajo) que tanto César como los que le ha ayudado con su contribución han invertido en esta guía. Por mi parte, quiero agradecer su trabajo y esfuerzo y reconocer el alto grado de calidad que tiene el producto resultante. Y estoy seguro de que el lector sabrá agradecerlo también sacando el mayor provecho de esta guía en sus nuevos retos de desarrollo.

Prólogo de José Murillo

(*Developer Solution Specialist, Microsoft DPE*)

Los grandes proyectos de software empresariales fracasan habitualmente. Es una afirmación dura, pero admitámoslo, es la cruda realidad con lo que todos los que llevamos años en el mundo del desarrollo de aplicaciones estamos familiarizados.

La "industria" del desarrollo de software apenas tiene 60 años. Durante este tiempo hemos ido aprendiendo a pasar de la arena al ladrillo, del ladrillo a los bloques prefabricados, pero todas estas técnicas de construcción perfectamente válidas para una casa son insuficientes e inútiles para grandes edificaciones. Si intentamos aplicarlas para estos macro-proyectos, el tiempo de desarrollo se multiplica exponencialmente o el edificio se derrumba al primer temblor o prueba de carga de los usuarios.

¿Qué está fallando? Para mí no hay ninguna duda, Gestión del Ciclo de Vida del Desarrollo y **Arquitectura Empresarial de Aplicaciones**. Tan importante como en la Arquitectura tradicional son el diseño, las estructuras y los cálculos de carga, en el mundo del desarrollo de software lo es la Arquitectura Software y de Sistemas. Es la disciplina que nos enseña como tenemos que combinar los bloques y tecnologías existentes para formar aplicaciones sólidas y duraderas. Este rol por desgracia está muy poco presente en las empresas actuales, donde cualquier buen programador con el paso del tiempo y una vez hay que reconocerle sus méritos pasados, es promocionado a "Jefe de Proyectos". ¿Pero qué demonios tiene que ver una cosa con la otra?

Este libro ofrece justamente las pautas, guías, recomendaciones y buenas prácticas para que los Arquitectos Software puedan diseñar aplicaciones empresariales sin reinventar la rueda, utilizando patrones existentes y buenas prácticas comprobadas. Es capaz de aterrizar con efectividad conceptos abstractos y multitud de las últimas tecnologías Microsoft en recomendaciones concretas para esos nuevos Arquitectos .NET.

De aquí mi reconocimiento y gracias por su trabajo a mi compañero y amigo Cesar de la Torre. Conozco perfectamente el gran esfuerzo personal que ha realizado para hacer realidad este proyecto, que estoy convencido repercutirá en la mejora de la calidad de las aplicaciones empresariales que se pongan en marcha siguiendo sus recomendaciones. Igualmente gracias al resto de colaboradores sin cuya ayuda este libro hubiese acabado con Cesar.

Prologo de Aurelio Porras
(*Developer Solution Specialist, Microsoft DPE*)

He tenido la oportunidad de participar en el desarrollo de alguna que otra aplicación de cierta envergadura y recuerdo gratamente esas reuniones en los inicios de los proyectos donde esbozábamos con cajas y flechas el esqueleto arquitectónico, detectábamos patrones y etiquetábamos cualquier elemento del diagrama con las últimas tecnologías disponibles que nos ayudaran a implementar de la mejor forma posible la funcionalidad requerida sin tener que reinventar la rueda. En esas discusiones arquitectónicas solían aflorar los típicos enfrentamientos sobre el nivel de complejidad de la arquitectura de la aplicación que se quería implementar: por un lado los partidarios de montar una arquitectura más sencilla, aprovechando bibliotecas de código e implementaciones de patrones ya construidas, para producir lógica de negocio enseguida y presentar resultados lo antes posible, dando más libertad al desarrollador a la hora de emplear las tecnologías; y por el otro los partidarios de construir una arquitectura más compleja, construyendo bibliotecas e implementando patrones a medida de la aplicación, para acelerar la producción de la lógica de negocio más adelante aunque se presentaran resultados más tarde, elevando el nivel de abstracción para evitar que el desarrollador tuviese que tomar decisiones tecnológicas. Era interesante ver cómo los "simplistas" increpaban los "complicados" el esfuerzo malgastado al construir arcos de iglesia innecesarios que los fabricantes de la infraestructura tecnológica en su siguiente versión harían obsoleta y el hastío que producían al desarrollador de la lógica de negocio que en ocasiones dejaba de ser un programador y se convertía en un mero configurador de la arquitectura; y los "complicados" reprendían a los "simplistas" por la cantidad de código duplicado que tiraban a la basura y el esfuerzo en coordinación que malgastaban para evitar esos problemas de duplicidad funcional al haber dado tanta libertad al desarrollador. Sí, suena al abuelo Cebolleta contando batallitas, pero es que era reuniones muy entretenidas.

El resultado final de esas discusiones y de algunas cañitas era una serie de decisiones arquitectónicas que determinaban la infraestructura tecnológica que se emplearía para construir la aplicación, las relaciones con sistemas externos, la organización del código en capas, las bibliotecas ya disponibles a usar y las que habría que desarrollar a medida, entre otras cosas. Recuerdo particularmente cómo tratábamos de desacoplar partes de la aplicación para facilitar su futura evolución, hasta donde el estado del arte de la tecnología nos dejaba llegar por aquel entonces, para poder modificar o extender la lógica de negocio sin tener que tocar todos los módulos o poder intercambiar uno de los sistemas externos, el servidor de aplicaciones o la base de datos sin muchos problemas.

Pero esas decisiones arquitectónicas no sólo estaban condicionadas por factores técnicos como las infraestructuras tecnológicas, los lenguajes de programación o las herramientas de desarrollo; sino también por factores propiamente relacionados con el desarrollo de un proyecto software como su presupuesto y duración, sus hitos y entregables, la experiencia del equipo de desarrollo, el conocimiento del negocio y aquellos porque-síes que tienen todos los proyectos. Al final la arquitectura podía sufrir esos indeseados tijeretazos por "decisiones de proyecto".

Pues bien, lamentablemente, también he tenido la oportunidad de comprobar cómo determinadas decisiones arquitectónicas pueden condenar el futuro de grandes aplicaciones.

Conozco el caso de una aplicación financiera que logra adaptarse a los cambios del negocio muy rápidamente gracias al alto nivel de abstracción que proporciona su arquitectura; el propio usuario es capaz de modificar la lógica de la aplicación a través de una herramienta visual y programando en un pseudo-lenguaje de negocio; el problema es que la capacidad de integración en línea con otros sistemas está muy limitada porque está construida sobre tecnologías obsoletas y su acoplamiento con éstas es tal, que el coste de migración a últimas tecnologías es demasiado alto y no se puede justificar desde el punto de vista del negocio; especialmente si tenemos en cuenta que la aplicación sigue funcionando como un reloj suizo y, siguiendo la máxima de esta nuestra industria, si funciona no lo toques. También conozco otra aplicación financiera bien desacoplada del servidor de aplicaciones y de la base de datos y que resulta relativamente sencillo actualizar tecnológicamente, pero que no cuidó la organización del código y la lógica de negocio está tan intrincada en las diferentes capas de la aplicación que no resulta tan ágil adaptarla a los cambios como al negocio le gustaría, y agilizarla supondría reescribir las tres cuartas partes de la aplicación; impensable, casi un nuevo proyecto. Seguramente las dos aplicaciones se idearon así por las circunstancias particulares que rodeaban a sus respectivos proyectos, pero está claro que las decisiones arquitectónicas tomadas en su momento han afectado negativamente al mantenimiento evolutivo de esas dos aplicaciones, que, como ya se preveía desde un principio, tendrían una larga duración en el entorno de producción.

Ésta es la madre del cordero que ha motivado esta guía. Naturalmente el estado del arte de la tecnología ha cambiado bastante, las tendencias arquitectónicas, las capacidades de las infraestructuras tecnológicas modernas, las novedades en los lenguajes de programación y las nuevas herramientas de desarrollo ayudan mucho a construir arquitecturas débilmente acopladas para facilitar el mantenimiento evolutivo de las aplicaciones; pero si además concebimos la arquitectura de la aplicación teniendo presente en primer lugar la importancia de su futura evolución, para adaptarse con facilidad a los cambios de negocio y para incorporar las últimas tecnologías sustituyendo a las que van quedando anticuadas, estaremos cerca de construir una gran aplicación de negocio con garantías de una vida saludable.

Y en esto ahonda la guía, en construir una arquitectura que desacople la lógica del negocio de la tecnología utilizada para construir la aplicación de forma que puedan evolucionar independientemente la una de la otra. Y no sólo habla de los pájaros y las flores, sino que se remanga a un nivel de detalle técnico que nos ilustrará en las últimas tecnologías .NET y herramientas de desarrollo de Microsoft y su aplicación en grandes aplicaciones de negocio, indicando cuándo usar qué tecnología y porqué, e incluyendo además el código de una aplicación de ejemplo siguiendo los preceptos indicados a lo largo la guía.

Por todo este material esclarecedor, agradezco a César el esfuerzo que ha realizado liderando esta iniciativa que seguro ayudará a arquitectos y desarrolladores a plantear arquitecturas de aplicaciones con una visión más holística, y extiendo el agradecimiento a los autores y los colaboradores que han participado en su elaboración. Enhorabuena por el resultado.

Israel Garcia Mesa
(*Consultor - Microsoft Services*)

Actualmente disponemos de un amplio abanico de opciones tecnológicas que podemos usar en nuestros proyectos y que cubren muchas necesidades que se han ido detectando a lo largo de los años. La experiencia que tenemos en *Microsoft Ibérica* es que esta variedad de alternativas no resuelve toda la problemática de los proyectos en nuestros clientes. El análisis que hemos realizado y que continuamos realizando para mejorar día a día nos ha proporcionado una serie de conclusiones que queremos compartir en esta guía.

Reflexiones de Arquitectura

El desarrollo de un proyecto de construcción de software es un proceso en el que intervienen muchos factores y por ello es importante contar con las herramientas adecuadas. Actualmente hay disponibles muchas opciones tecnológicas que nos ayudan a componer nuestras soluciones, pero sin embargo no mitigan las principales problemáticas de un proyecto:

- Necesidades de adaptación a cambios en los proyectos (requerimientos funcionales y técnicos), que pueden tener un alto impacto en lo que a esfuerzo se refiere.
- Incertidumbre a la hora de escoger y utilizar la tecnología que mejor encaja en cada escenario.
- Integración con sistemas heredados que no tienen un alineamiento claro con los requerimientos de proyecto.

Estas y otras situaciones pueden afectar al desarrollo de los proyectos y aumentar la posibilidad de que se manifiesten nuevos riesgos que impacten al proyecto. Con el fin de mitigar estos riesgos es recomendable:

- La metodología de trabajo debe adaptarse a nuestro equipo, a nuestro tipo de proyecto y a nuestro cliente, puesto que será nuestra táctica para alcanzar nuestro objetivo y hay que tener en cuenta todos los detalles. Por tanto, es importante escoger un método de trabajo adaptado al contexto del proyecto en donde hay que considerar el tipo de solución y el equipo de trabajo.
- Considerar un modelo de arquitectura que satisfaga las necesidades conocidas y con un bajo nivel de acoplamiento, lo que facilitará su adaptabilidad. En este punto pueden elegirse distintas opciones a la hora de plantear el sistema, pero seguir el modelo plantado por el Diseño Dirigido al Dominio (DDD) nos puede ayudar a seguir el planteamiento más adecuado.

El diseño de una solución, aparte de ser un proceso incremental, puede ser un proceso a realizar desde distintos enfoques hasta completar la visión de la solución. De la experiencia

recogida en los distintos proyectos que nos hemos desarrollado, hemos visto útiles algunos planteamientos que resumimos a continuación:

- Las soluciones, sean del tamaño que sean, nacen de un diseño global en donde los aspectos técnicos no son relevantes (podríamos hablar de diseño conceptual) y posteriormente diseñar las partes de la solución a medida que nos tengamos que ir enfocando en cada una de ellas. Con este modelo poco a poco nos iremos acercando a los detalles de la implementación desacoplando el diseño, reduciendo la complejidad y la posibilidad de que un problema técnico pueda afectar al resto de la solución.
- Así mismo, será necesario conjugar el diseño del modelo lógico con el o los modelos físicos, siendo lo ideal que un planteamiento condicione lo menos posible al otro. Este tipo de planteamientos facilita la reutilización y la adaptabilidad de la solución a distintos escenarios.

Siempre estará la tentación de construir la solución entorno a la idea de que la tecnología resolverá nuestros problemas, y nos parecerá que es un camino corto a nuestros objetivos. Sin embargo, podemos descubrir que no es el camino más rápido ya que cuando un diseño no puede crecer y/o evolucionar porque o bien nos requiere un alto esfuerzo o no controlamos el impacto de dichos cambios, entonces es cuando la tecnología no aporta valor a la solución y puede convertirse en un problema.

Adicionalmente hay una serie de herramientas muy útiles a la hora de construir una solución y que nos ayudan también a la hora de abordar cambios en la implementación y en el diseño de la misma:

- **Desarrollo de Pruebas:** disponer de pruebas unitarias y funcionales automatizadas nos ayudará a conocer la estabilidad de nuestra solución, y por lo tanto determinar si algún cambio ha podido afectar a la solución y en qué punto.
- **Refactorización:** plantear e implementar cambios en la solución mediante técnicas de *refactoring* es una manera eficiente que nos ayuda a controlar el impacto de los mismos. Complementar la refactorización con el uso de pruebas ayuda a reducir riesgos, por lo que son dos herramientas perfectamente complementarias.
- **Comunicación:** una buena comunicación dentro del equipo, reduce la posibilidad de trabajar de manera ineficiente o incluso duplicar funcionalidad. Además es un instrumento útil en nuestra relación con el cliente ayudándonos a poner en común expectativas, detectar nuevos requerimientos o posibles riesgos rápida y ágilmente.

Estas conclusiones que pueden parecer lógicas y sin embargo difíciles de llevar a cabo, son la razón por la queremos compartir el conocimiento presente en esta guía con el fin de que nuestra experiencia pueda ser útil en los proyectos y la tecnología se convierta en esa **herramienta que hace más fácil nuestro trabajo**.

Arquitectura Marco .NET
Microsoft Ibérica

1.- INTRODUCCIÓN

Microsoft Ibérica ha detectado en múltiples clientes y *partners* la necesidad de disponer de una "Guía de Arquitectura base .NET" en español, que sirva para marcar unas líneas maestras de diseño e implementación a la hora de desarrollar aplicaciones .NET <u>complejas y con una vida y evolución de larga duración</u>. Este marco de trabajo común (en muchas empresas denominado "Libro Blanco") define un camino para diseñar e implementar aplicaciones empresariales de envergadura, con un volumen importante de lógica de negocio. Seguir estas guías ofrece importantes beneficios en cuanto a calidad, estabilidad y, especialmente, un incremento en la facilidad del mantenimiento futuro de las aplicaciones, debido al desacoplamiento entre sus componentes, así como por la homogeneidad y similitudes de los diferentes desarrollos a realizar.

Microsoft Ibérica define el presente '***Libro de Arquitectura Marco***' como patrón y modelo base, sin embargo, en ningún caso este marco debe ser inalterable. Al contrario, se trata del primer peldaño de una escalera, un acelerador inicial, que debería ser personalizado y modificado por cada organización que lo adopte, enfocándolo hacia necesidades concretas, adaptándolo y agregándole funcionalidad específica según el mercado objetivo, etc.

1.1.- Audiencia del documento

Este documento está dirigido a las personas involucradas en todo el ciclo de vida de productos *software* o de aplicaciones corporativas desarrolladas a medida. Especialmente los siguientes perfiles:

- Arquitecto de Software
- Desarrollador

1.2.- Objetivos de la Arquitectura Marco .NET

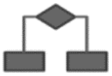
Este documento pretende describir una arquitectura marco sobre la que desarrollar las aplicaciones a medida y establece un conjunto de normas, mejores prácticas y guías de desarrollo para utilizar .NET de forma adecuada y, sobre todo, homogénea.

***DESCARGO DE RESPONSABILIDAD*:**
Queremos insistir en este punto y destacar que la presente propuesta de '***Arquitectura N-Capas Orientada al Dominio***' no es adecuada para cualquier tipo de aplicaciones, **solamente es adecuada para aplicaciones complejas empresariales con un volumen importante de lógica de negocio y una vida y evolución de aplicación de larga duración**, donde es importante implementar conceptos de desacoplamiento y ciertos patrones DDD. Para aplicaciones pequeñas y orientadas a datos, probablemente sea más adecuada una aproximación de arquitectura más sencilla implementada con tecnologías RAD.
Así mismo, esta guía (y su aplicación ejemplo asociada) es simplemente una propuesta a tener en cuenta y ser evaluada y personalizada por las organizaciones y empresas que lo deseen. *Microsoft Ibérica* no se hace responsable de problemas que pudieran derivarse de ella.

1.3.- Niveles de la documentación de la Arquitectura marco .NET

La documentación de esta arquitectura se diseña en dos niveles principales:

- **Nivel lógico de Arquitectura de Software:** Este primer nivel lógico, es una Arquitectura de software agnóstica a la tecnología, donde no se especifican tecnologías concretas de .NET. Para resaltar este nivel, se mostrará el icono:



- **Nivel de Implementación de Arquitectura .NET:** Este segundo nivel, es la implementación concreta de Arquitectura .NET, donde se enumerarán las tecnologías posibles para cada escenario con versiones concretas; normalmente se escogerá una opción y se explicará su implementación. Así mismo, la implementación de la arquitectura cuenta con una aplicación .NET ejemplo, cuyo alcance funcional es muy pequeño, pero debe implementar todas y cada una de las áreas tecnológicas de la Arquitectura marco. Para resaltar este nivel, se mostrará el icono de .NET al inicio del capítulo:

1.4.- Aplicación Ejemplo en CODEPLEX

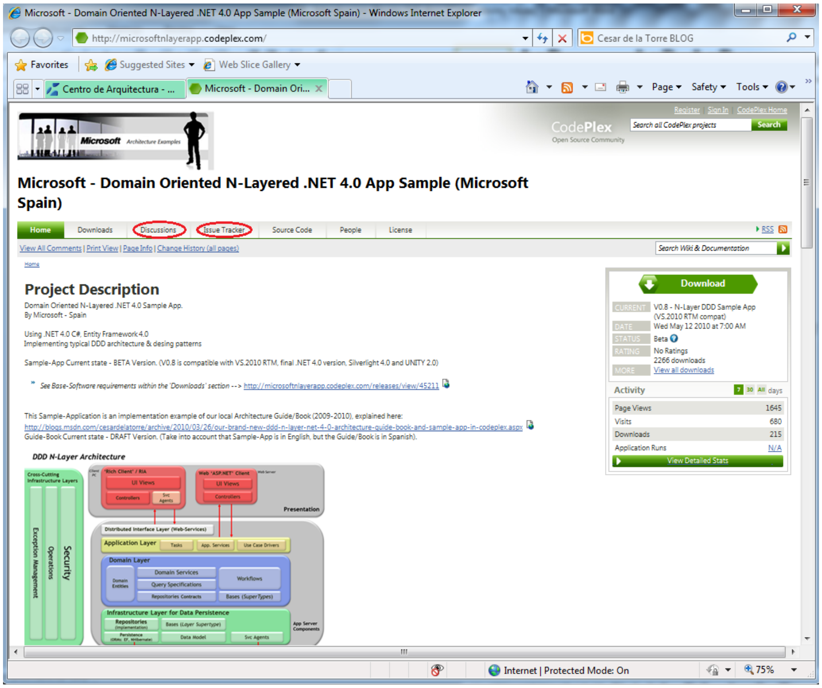
Es fundamental destacar que simultáneamente a la elaboración de este libro/guía de Arquitectura, también hemos desarrollado una aplicación ejemplo, que implementa los patrones expuestos en esta guía, con las últimas tecnologías actuales de Microsoft ('Ola .NET 4.0').

Así mismo, la mayoría de los *snippets* de código mostrados en este libro, son extractos de código precisamente de esta Aplicación ejemplo.

Esta aplicación ejemplo está publicada en CODEPLEX como código OSS y se puede descargar desde la siguiente URL:

CodePlex Open Source Project Community	http://microsoftnlayerapp.codeplex.com/

En CODEPLEX disponemos no solo del código fuente de la aplicación ejemplo, también de cierta documentación sobre requerimientos (tecnologías necesarias como *Unity 2.0, PEX & MOLES, WPF Toolkit, Silverlight 4 Tools for Visual Studio 2010, Silverlight 4.0 Toolkit, AppFabric*, etc., links desde donde descargarlas en Internet, etc.) y de una página de **Discusiones/Foro**, algo muy interesante para poder colaborar con la comunidad, y poder también presentarnos preguntas, ideas, propuestas de evolución, etc.:



La aplicación ejemplo implementa los diferentes patrones de Diseño y Arquitectura DDD, pero con las últimas tecnologías Microsoft. También dispone de varios clientes (WPF, Silverlight, ASP.NET MVC) y otros a ser añadidos como OBA y Windows Phone 7.0, etc.

Es importante resaltar que la funcionalidad de la aplicación ejemplo, es lógicamente, bastante sencilla, pues lo que se quiere resaltar es la Arquitectura, no implementar un volumen grande de funcionalidad que complique el seguimiento y entendimiento de la Arquitectura.

La Capa de presentación y las diferentes implementaciones son simplemente un área más de la arquitectura y no son precisamente 'el *core*' de esta guía de referencia, donde nos centramos más en capas relativas al servidor de componentes (Capa del Dominio, de Aplicación, Infraestructura de acceso a datos, son sus respectivos patrones). Aun así, se hace también una revisión de los diferentes patrones en capa de presentación (MVC, M-V-VM, etc.) y como implementarlos con diferentes tecnologías.

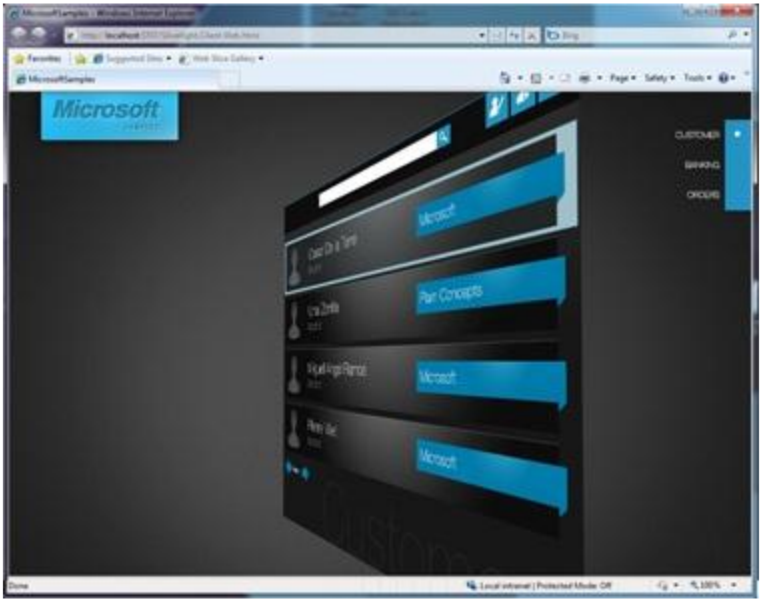
Aquí mostramos algunas pantallas capturadas de la aplicación ejemplo:

Cliente Silverlight 4.0

Silverlight – Lista de Clientes



Silverlight – Transición de Silverlight

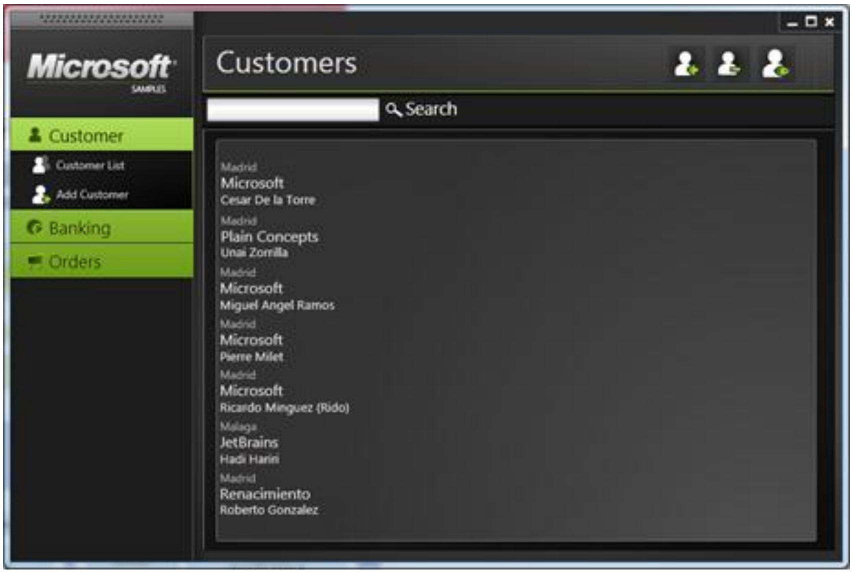


Silverlight – 'Vista de Cliente'



Cliente WPF 4.0

WPF – Vista de ‘Lista de Clientes’



WPF – Vista de Cliente

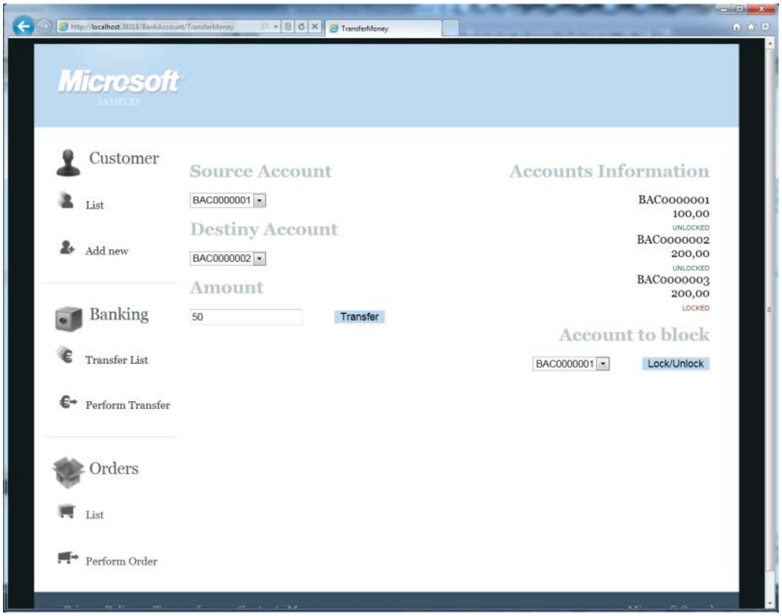


WPF – 'Transferencias Bancarias'

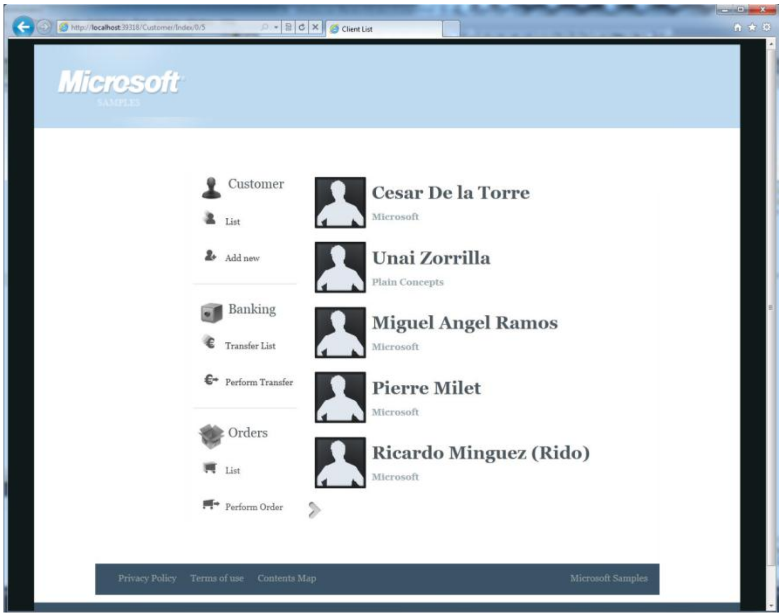


Cliente ASP.NET MVC

MVC – 'Transferencias Bancarias'



MVC – Vista de 'Lista de Clientes'



Por último, resaltar que tanto la aplicación como todo el código fuente e dicha aplicación, lo hemos elaborado en inglés, para poder ser aprovechada por toda la comunidad, a nivel mundial y no solo en Español.

Recomendamos 'bajar' de Internet esta aplicación ejemplo e irla investigando en paralelo según se lee la presente guía/libro de Arquitectura, especialmente cuando se está leyendo los apartados de implementación marcados con el siguiente logo de .NET:

CAPÍTULO

1

Fundamentos de Arquitectura de Aplicaciones

El diseño de la arquitectura de un sistema es el proceso por el cual se define una solución para los requisitos técnicos y operacionales del mismo. Este proceso define qué componentes forman el sistema, cómo se relacionan entre ellos, y cómo mediante su interacción llevan a cabo la funcionalidad especificada, cumpliendo con los criterios de calidad indicados como seguridad, disponibilidad, eficiencia o usabilidad.

Durante el diseño de la arquitectura se tratan los temas que pueden tener un impacto importante en el éxito o fracaso de nuestro sistema. Algunas preguntas que hay que hacerse al respecto son:

- ¿En qué entorno va a ser desplegado nuestro sistema?
- ¿Cómo va a ser nuestro sistema puesto en producción?
- ¿Cómo van a utilizar los usuarios nuestro sistema?
- ¿Qué otros requisitos debe cumplir el sistema? (seguridad, rendimiento, concurrencia, configuración…)
- ¿Qué cambios en la arquitectura pueden impactar al sistema ahora o una vez desplegado?

Para diseñar la arquitectura de un sistema es importante tener en cuenta los intereses de los distintos agentes que participan. Estos agentes son los usuarios del sistema, el propio sistema y los objetivos del negocio. Cada uno de ellos impone requisitos y restricciones que deben ser tenidos en cuenta en el diseño de la arquitectura y que pueden llegar a entrar en conflicto, por lo que se debe alcanzar un compromiso entre los intereses de cada participante.

Para los usuarios es importante que el sistema responda a la interacción de una forma fluida, mientras que para los objetivos del negocio es importante que el sistema

cueste poco. Los usuarios pueden querer que se implemente primero una funcionalidad útil para su trabajo, mientras que el sistema puede tener prioridad en que se implemente la funcionalidad que permita definir su estructura.

El trabajo del arquitecto es delinear los escenarios y requisitos de calidad importantes para cada agente así como los puntos clave que debe cumplir y las acciones o situaciones que no deben ocurrir.

El objetivo final de la arquitectura es identificar los requisitos que producen un impacto en la estructura del sistema y reducir los riesgos asociados con la construcción del sistema. La arquitectura debe soportar los cambios futuros del software, del hardware y de funcionalidad demandada por los clientes. Del mismo modo, es responsabilidad del arquitecto analizar el impacto de sus decisiones de diseño y establecer un compromiso entre los diferentes requisitos de calidad así como entre los compromisos necesarios para satisfacer a los usuarios, al sistema y los objetivos del negocio.

En síntesis, la arquitectura debería:

- Mostrar la estructura del sistema pero ocultar los detalles.
- Realizar todos los casos de uso.
- Satisfacer en la medida de lo posible los intereses de los agentes.
- Ocuparse de los requisitos funcionales y de calidad.
- Determinar el tipo de sistema a desarrollar.
- Determinar los estilos arquitecturales que se usarán.
- Tratar las principales cuestiones transversales.

Una vez vistas las principales cuestiones que debe abordar el diseño de la arquitectura del sistema, ahora vamos a ver los pasos que deben seguirse para realizarlo. En una metodología ágil como *Scrum*, la fase de diseño de la arquitectura comienza durante en el pre-juego (Pre-game) o en la fase de Inicio (*Inception*) en RUP, en un punto donde ya hemos capturado la visión del sistema que queremos construir. En el diseño de la arquitectura lo primero que se decide es el tipo de sistema o aplicación que vamos a construir. Los principales tipos son aplicaciones móviles, de escritorio, RIAs (*Rich Internet Application*), aplicaciones de servicios, aplicaciones web… Es importante entender que el tipo de aplicación viene determinado por la topología de despliegue y los requisitos y restricciones indicadas en los requisitos.

La selección de un tipo de aplicación determina en cierta medida el estilo arquitectural que se va a usar. El estilo arquitectural es en esencia la partición más básica del sistema en bloques y la forma en que se relacionan estos bloques. Los principales estilos arquitecturales son Cliente/Servidor, Sistemas de Componentes, Arquitectura en capas, MVC, N-Niveles, SOA… Como ya hemos dicho, el estilo

arquitectural que elegimos depende del tipo de aplicación. Una aplicación que ofrece servicios lo normal es que se haga con un estilo arquitectural SOA.

Por otra parte, a la hora de diseñar la arquitectura tenemos que entender también que un tipo de aplicación suele responder a más de un estilo arquitectural. Por ejemplo, una página web hecha con ASP.NET MVC sigue un estilo Cliente/Servidor pero al mismo tiempo el servidor sigue un estilo Modelo Vista Controlador.

Tras haber seleccionado el tipo de aplicación y haber determinado los estilos arquitecturales que más se ajustan al tipo de sistema que vamos a construir, tenemos que decidir cómo vamos a construir los bloques que forman nuestro sistema. Por ello el siguiente paso es seleccionar las distintas tecnologías que vamos a usar. Estas tecnologías están limitadas por las restricciones de despliegue y las impuestas por el cliente. Hay que entender las tecnologías como los ladrillos que usamos para construir nuestro sistema. Por ejemplo, para hacer una aplicación web podemos usar la tecnología ASP.NET o para hacer un sistema que ofrece servicios podemos emplear WCF.

Cuando ya hemos analizado nuestro sistema y lo hemos fragmentado en partes más manejables, tenemos que pensar como implementamos todos los requisitos de calidad que tiene que satisfacer. Los requisitos de calidad son las propiedades no funcionales que debe tener el sistema, como por ejemplo la seguridad, la persistencia, la usabilidad, la mantenibilidad, etc. Conseguir que nuestro sistema tenga estas propiedades va a traducirse en implementar funcionalidad extra, pero esta funcionalidad es ortogonal a la funcionalidad básica del sistema.

Para tratar los requisitos de calidad el primer paso es preguntarse ¿Qué requisitos de calidad requiere el sistema? Para averiguarlo tenemos que analizar los casos de uso. Una vez hemos obtenido un listado de los requisitos de calidad las siguientes preguntas son ¿Cómo consigo que mi sistema cumpla estos requisitos? ¿Se puede medir esto de alguna forma? ¿Qué criterios indican que mi sistema cumple dichos requisitos?

Los requisitos de calidad nos van a obligar a tomar decisiones transversales sobre nuestro sistema. Por ejemplo, cuando estamos tratando la seguridad de nuestro sistema tendremos que decidir cómo se autentican los usuarios, como se maneja la autorización entre las distintas capas, etc. De la misma forma tendremos que tratar otros temas como las comunicaciones, la gestión de excepciones, la instrumentación o el cacheo de datos.

Los procesos software actuales asumen que el sistema cambiará con el paso del tiempo y que no podemos saber todo a la hora de diseñar la arquitectura. El sistema tendrá que evolucionar a medida que se prueba la arquitectura contra los requisitos del mundo real. Por eso, no hay que tratar de formalizar absolutamente todo a la hora de definir la arquitectura del sistema. Lo mejor es no asumir nada que no se pueda comprobar y dejar abierta la opción de un cambio futuro. No obstante, sí que existirán algunos aspectos que podrán requerir un esfuerzo a la hora de realizar modificaciones. Para minimizar dichos esfuerzos es especialmente importante el concepto de desacoplamiento entre componentes. Por ello es vital identificar esas partes de nuestro sistema y detenerse el tiempo suficiente para tomar la decisión correcta. En síntesis las claves son:

- Construir 'hasta el cambio' más que 'hasta el final'.
- Utilizar herramientas de modelado para analizar y reducir los riesgos.
- Utilizar modelos visuales como herramienta de comunicación.
- Identificar las decisiones clave a tomar.

A la hora de crear la arquitectura de nuestro sistema de forma iterativa e incremental, las principales preguntas a responder son:

- ¿Qué partes clave de la arquitectura representan el mayor riesgo si las diseño mal?
- ¿Qué partes de la arquitectura son más susceptibles de cambiar?
- ¿Qué partes de la arquitectura puedo dejar para el final sin que ello impacte en el desarrollo del sistema?
- ¿Cuáles son las principales suposiciones que hago sobre la arquitectura y como las verifico?
- ¿Qué condiciones pueden provocar que tenga que cambiar el diseño?

Como ya hemos dicho, los procesos modernos se basan en adaptarse a los cambios en los requisitos del sistema y en ir desarrollando la funcionalidad poco a poco**.** En el plano del diseño de la arquitectura, esto se traduce en que **definiremos la arquitectura del sistema final poco a poco.** Podemos entenderlo como un proceso de maduración, como el de un ser vivo**. Primero tendremos una arquitectura a la que llamaremos línea base y que es una visión del sistema en el momento actual del proceso. Junto a esta línea base tendremos una serie de arquitecturas candidatas que serán el siguiente paso en la maduración de la arquitectura.** Cada arquitectura candidata incluye el tipo de aplicación, la arquitectura de despliegue, el estilo arquitectural, las tecnologías seleccionadas, los requisitos de calidad y las decisiones transversales. Las preguntas que deben responder las arquitecturas candidatas son:

- ¿Qué suposiciones he realizado en esta arquitectura?
- ¿Qué requisitos explícitos o implícitos cumple esta arquitectura?
- ¿Cuáles son los riesgos tomados con esta evolución de la arquitectura?
- ¿Qué medidas puedo tomar para mitigar esos riesgos?
- ¿En qué medida esta arquitectura es una mejora sobre la línea base o las otras arquitecturas candidatas?

Dado que usamos una metodología iterativa e incremental para el desarrollo de nuestra arquitectura, la implementación de la misma debe seguir el mismo patrón. La forma de hacer esto es mediante pruebas arquitecturales. Estas pruebas son pequeños desarrollos de parte de la aplicación (Pruebas de Concepto) que se usan para mitigar riesgos rápidamente o probar posibles vías de maduración de la arquitectura. Una prueba arquitectural se convierte en una arquitectura candidata que se evalúa contra la línea base. Si es una mejora, se convierte en la nueva línea base frente a la cual crear y evaluar las nuevas arquitecturas candidatas. Las preguntas que debemos hacerle a una arquitectura candidata que surge como resultado de desarrollar una prueba arquitectural son:

- ¿Introduce nuevos riesgos?
- ¿Soluciona algún riesgo conocido esta arquitectura?
- ¿Cumple con nuevos requisitos del sistema?
- ¿Realiza casos de uso arquitecturalmente significativos?
- ¿Se encarga de implementar algún requisito de calidad?
- ¿Se encarga de implementar alguna parte del sistema transversal?

Los casos de uso importantes son aquellos que son críticos para la aceptación de la aplicación o que desarrollan el diseño lo suficiente como para ser útiles en la evaluación de la arquitectura.

En resumen, el proceso de diseño de la arquitectura tiene que decidir qué funcionalidad es la más importante a desarrollar. A partir de esta decisión tiene que decidir el tipo de aplicación y el estilo arquitectural, y tomar las decisiones importantes sobre seguridad, rendimiento… que afectan al conjunto del sistema. El diseño de la arquitectura decide cuales son los componentes más básicos del sistema y como se relacionan entre ellos para implementar la funcionalidad. Todo este proceso debe hacerse paso a paso, tomando solo las decisiones que se puedan comprobar y dejando abiertas las que no. Esto significa mitigar los riesgos rápidamente y explorar la implementación de casos de uso que definan la arquitectura.

CAPÍTULO

2

El proceso de Diseño de la Arquitectura

En el marco de la ingeniería del software y del ALM, el proceso de diseño de la arquitectura juega un papel muy importante. La diferencia entre un buen proceso de diseño arquitectural y uno malo puede suponer la diferencia entre el fracaso o éxito de nuestro proyecto. En el diseño de la arquitectura tratamos los temas más importantes a la hora de definir nuestro sistema, es decir, creamos un molde básico de nuestra aplicación. Dentro del proceso de diseño de la arquitectura se decide:

- Qué tipo de aplicación se va a construir. (Web, RIA, Rich Client…)
- Qué estructura lógica va a tener la aplicación (N-Capas, Componentes…)
- Qué estructura física va a tener la aplicación (Cliente/Servidor, N-Tier…)
- Qué riesgos hay que afrontar y cómo hacerlo. (Seguridad, Rendimiento, Flexibilidad…)
- Qué tecnologías vamos a usar (WCF,WF,WPF, Silverlight, Entity Framework, etc.)

Para realizar todo este proceso partiremos de la información que ha generado el proceso de captura de requisitos, más detalladamente, esta información es:

- Casos de uso o historias de usuario.
- Requisitos funcionales y no funcionales.
- Restricciones tecnológicas y de diseño en general.

- Entorno de despliegue propuesto.

A partir de esta información deberemos generar los artefactos necesarios para que los programadores puedan implementar correctamente el sistema. Como mínimo, en el proceso de diseño de la arquitectura debemos definir:

- Casos de uso significativos a implementar.
- Riesgos a mitigar y cómo hacerlo.
- Arquitecturas candidatas a implementar.

Como ya hemos dicho, el diseño de la arquitectura es un proceso iterativo e incremental. En el diseño de la arquitectura repetimos 5 pasos hasta completar el desarrollo del sistema completo. Los pasos que repetimos y la forma más clara de verlos es esta:



Figura 1.- Diseño de Arquitectura

A continuación vamos a examinar en más detalle cada uno de estos pasos para comprender qué debemos definir y dejar claro en cada uno de ellos.

1.- IDENTIFICAR LOS OBJETIVOS DE LA ITERACIÓN

Los objetivos de la iteración son el primer paso para dar forma a la arquitectura de nuestro sistema. En este punto lo importante es analizar las restricciones que tiene nuestro sistema en cuanto a tecnologías, topología de despliegue, uso del sistema, etc… En esta fase es muy importante marcar cuales van a ser los objetivos de la arquitectura, tenemos que decidir si estamos construyendo un prototipo, realizando un diseño completo o probando posibles vías de desarrollo de la arquitectura. También hay que tener en cuenta en este punto a las personas que forman nuestro equipo. El tipo de documentación a generar así como el formato dependerá de si nos dirigimos a otros arquitectos, a desarrolladores, o a personas sin conocimientos técnicos.

El objetivo de esta fase del proceso de diseño de la arquitectura es entender por completo el entorno que rodea a nuestro sistema. Esto nos permitirá decidir en qué centraremos nuestra actividad en las siguientes fases del diseño y determinará el alcance y el tiempo necesarios para completar el desarrollo. Al término de esta fase deberemos tener una lista de los objetivos de la iteración, preferiblemente con planes para afrontarlos y métricas para determinar el tiempo y esfuerzo que requerirá completarlos. Tras esta fase es imprescindible tener una estimación del tiempo que invertiremos en el resto del proceso.

2.- SELECCIONAR LOS CASOS DE USO ARQUITECTURALMENTE IMPORTANTES

El diseño de la arquitectura es un proceso dirigido por el cliente y los riesgos a afrontar, esto significa que desarrollaremos primero los casos de uso (funcionalidad) que más valor tengan para el cliente y mitigaremos en primer lugar los riesgos más importantes que afronte nuestra arquitectura (requisitos de calidad). La importancia de un caso de uso la valoraremos según los siguientes criterios:

- Lo importante que es el caso de uso dentro de la lógica de negocio: Esto vendrá dado por la frecuencia de utilización que tendrá el caso de uso en el sistema en producción o el valor que aporte esa funcionalidad al cliente.

- El desarrollo del caso de uso implica un desarrollo importante de la arquitectura: Si el caso de uso afecta a todos los niveles de la arquitectura es un firme candidato a ser prioritario, ya que su desarrollo e implementación permitirán definir todos los niveles de la arquitectura aumentando la estabilidad de la misma.

- El desarrollo del caso de uso implica tratar algún requisito de calidad: Si el caso de uso requiere tratar temas como la seguridad, la disponibilidad o la tolerancia a fallos del sistema, es un caso de uso importante ya que permite tratar los aspectos horizontales del sistema a la vez que se desarrolla la funcionalidad.

- Lo que se adapte el caso de uso a los objetivos de la iteración: A la hora de seleccionar los casos de uso que vamos a implementar tenemos que tener en cuenta lo que se ajustan a los objetivos que nos hemos marcado para la iteración. No vamos a escoger casos de uso que desarrollen mucho el conjunto del sistema si nos hemos marcado como objetivo de la iteración reducir bugs o mitigar algún riesgo dado.

Es muy importante tener claro que no se debe tratar de diseñar la arquitectura del sistema en una sola iteración. En esta fase del proceso de diseño analizamos todos los casos de uso y seleccionamos solo un subconjunto, el más importante arquitecturalmente y procedemos a su desarrollo. En este punto, solo definimos los aspectos de la arquitectura que conciernen a los casos de uso que hemos seleccionado y dejamos abiertos el resto de aspectos para futuras iteraciones. Es importante recalcar que puede que en una iteración no definamos por completo algún aspecto del sistema, pero lo que tenemos que tener claro es que debemos intentar minimizar el número de cambios en futuras iteraciones. Esto no significa que no debamos "asumir que el software evoluciona", sino que cuando desarrollemos un aspecto del sistema no nos atemos a una solución específica sino que busquemos una solución genérica que permita afrontar los posibles cambios en futuras iteraciones. En definitiva, todo esto se resume en dar pasos cortos pero firmes.

Es interesante a la hora de desarrollar el sistema tener en cuenta las distintas historias de usuario, sistema y negocio. Las historias de usuario, sistema y negocio son pequeñas frases o párrafos que describen aspectos del sistema desde el punto de vista del implicado. Las historias de usuario definen como los usuarios utilizarán el sistema, las historias de sistema definen los requisitos que tendrá que cumplir el sistema y como se organizará internamente y las historias de negocio definen como el sistema cumplirá con las restricciones de negocio.

Desmenuzar los casos de uso en varias historias de usuario, sistema y negocio nos permitirá validar más fácilmente nuestra arquitectura asegurándonos de que cumple con las historias de usuario, sistema y negocio de la iteración.

3.- REALIZAR UN ESQUEMA DEL SISTEMA

Una vez que están claros los objetivos de la iteración y la funcionalidad que desarrollaremos, podemos pasar a su diseño. Llegados a este punto, el primer paso es decidir qué tipo de aplicación vamos a desarrollar. El tipo de aplicación que elegiremos dependerá de las restricciones de despliegue, de conectividad, de lo compleja que sea la interfaz de usuario y de las restricciones de interoperabilidad, flexibilidad y tecnologías que imponga el cliente. Cada tipo de aplicación nos ofrece una serie de ventajas e inconvenientes, el arquitecto tiene que escoger el tipo de aplicación que mejor se ajuste a las ventajas que espera que tenga su sistema y que presente menos inconvenientes. Los principales tipos de aplicaciones que desarrollaremos son:

- **Aplicaciones para dispositivos móviles:** Se trata de aplicaciones web con una interfaz adaptada para dispositivos móviles o aplicaciones de usuario desarrolladas para el terminal.
- **Aplicaciones de escritorio:** Son las aplicaciones clásicas que se instalan en el equipo del usuario que la vaya a utilizar.
- **RIA (Rich Internet Applications):** Se trata de aplicaciones que se ejecutan dentro del navegador gracias a un plug-in y que ofrecen una mejor respuesta que las aplicaciones web y una interfaz de calidad similar a las aplicaciones de usuario con la ventaja de que no hay que instalarlas.
- **Servicios:** Se trata de aplicaciones que exponen una funcionalidad determinada en forma de servicios web para que otras aplicaciones los consuman.
- **Aplicaciones web:** Son aplicaciones que se consumen mediante un navegador y que ofrecen una interfaz de usuario estándar y completamente interoperable.

A modo de resumen y guía, la siguiente tabla recoge las principales ventajas y consideraciones a tener en cuenta para cada tipo de aplicación:

Tabla 1.- Ventajas y consideraciones tipos de aplicación

Tipo de aplicación	Ventajas	Consideraciones
Aplicaciones para dispositivos móviles	• Sirven en escenarios sin conexión o con conexión limitada. • Se pueden llevar en dispositivos de mano. • Ofrecen alta disponibilidad y fácil acceso a los usuarios fuera de su entorno habitual.	• Limitaciones a la hora de interactuar con la aplicación. • Tamaño de la pantalla reducido.
Aplicaciones de escritorio	• Aprovechan mejor los recursos de los clientes. • Ofrecen la mejor respuesta a la interacción, una interfaz más potente y mejor experiencia de usuario.	• Despliegue complejo. • Versionado complicado. • Poca interoperabilidad.

	• Proporcionan una interacción muy dinámica. • Soportan escenarios desconectados o con conexión limitada.	
RIA (Rich Internet Applications)	• Proporcionan la misma potencia gráfica que las aplicaciones de escritorio. • Ofrecen soporte para visualizar contenido multimedia. • Despliegue y distribución simples.	• Algo más pesadas que las aplicaciones web. • Aprovechan peor los recursos que las aplicaciones de escritorio. • Requieren tener instalado un plugin para funcionar.
Aplicaciones orientadas a servicios	• Proporcionan una interfaz muy desacoplada entre cliente y servidor. • Pueden ser consumidas por varias aplicaciones sin relación. • Son altamente interoperables	• No tienen interfaz gráfica. • Necesitan conexión a internet.
Aplicaciones web	• Llegan a todo tipo de usuarios y tienen una interfaz de usuario estándar y multiplataforma. • Son fáciles de desplegar y de actualizar.	• Dependen de la conectividad a red. • No pueden ofrecer interfaces de usuario complejas.

Una vez que tenemos decidido el tipo de aplicación que vamos a desarrollar, el siguiente paso es diseñar la arquitectura de la infraestructura, es decir, la topología de despliegue. La topología de despliegue depende directamente de las restricciones impuestas por el cliente, de las necesidades de seguridad del sistema y de la infraestructura disponible para desplegar el sistema. Definimos la arquitectura de la infraestructura en este punto, para tenerla en consideración a la hora de diseñar la arquitectura lógica de nuestra aplicación. Dado que las capas son más "maleables" que los niveles, encajaremos las distintas capas lógicas dentro de los niveles del sistema. Generalizando existen dos posibilidades, despliegue distribuido y despliegue no distribuido.

El despliegue no distribuido tiene la ventaja de ser más simple y más eficiente en las comunicaciones ya que las llamadas son locales. Por otra parte, de esta forma es más difícil permitir que varias aplicaciones utilicen la misma lógica de negocio al mismo tiempo. Además en este tipo de despliegue los recursos de la máquina son compartidos por todas las capas con lo que si una capa emplea más recursos que las otras existirá un cuello de botella.

El despliegue distribuido permite separar las capas lógicas en distintos niveles físicos. De esta forma el sistema puede aumentar su capacidad añadiendo servidores donde se necesiten y se puede balancear la carga para maximizar la eficiencia. Al mismo tiempo, al separar las capas en distintos niveles aprovechamos mejor los recursos, balanceando el número de equipos por nivel en función del consumo de las capas que se encuentran en él. El lado malo de las arquitecturas distribuidas es que la serialización de la información y su envío por la red tienen un coste no despreciable. Así mismo, los sistemas distribuidos son más complejos y más caros.

Tras decidir qué tipo de aplicación desarrollaremos y cuál será su topología de despliegue llega el momento de diseñar la arquitectura lógica de la aplicación. Para ello emplearemos en la medida de lo posible un conjunto de estilos arquitecturales conocidos. Los estilos arquitecturales son "patrones" de nivel de aplicación que definen un aspecto del sistema que estamos diseñando y representan una forma estándar de definir o implementar dicho aspecto. La diferencia entre un estilo arquitectural y un patrón de diseño es el nivel de abstracción, es decir, un patrón de diseño da una especificación concreta de cómo organizar las clases y la interacción entre objetos, mientras que un estilo arquitectural da una serie de indicaciones sobre qué se debe y qué no se debe hacer en un determinado aspecto del sistema. Los estilos arquitecturales se pueden agrupar según el aspecto que definen como muestra la siguiente tabla:

Tabla 2.- Aspectos estilos estructurales

Aspecto	Estilos arquitecturales
Comunicaciones	SOA, Message Bus, Tuberías y filtros.
Despliegue	Cliente/Servidor, 3-Niveles, N-Niveles.
Dominio	Modelo de dominio, Repositorio.
Interacción	Presentación separada.
Estructura	Componentes, Orientada a objetos, Arquitectura en capas.

Como se desprende de la tabla, en una aplicación usaremos varios estilos arquitecturales para dar forma al sistema. Por tanto, una aplicación será una combinación de muchos de ellos y de soluciones propias.

Ahora que ya hemos decidido el tipo de aplicación, la infraestructura física y la estructura lógica, tenemos una buena idea del sistema que construiremos. El siguiente paso lógico es comenzar con la implementación del diseño y para ello lo primero que tenemos que hacer es decidir qué tecnologías emplearemos. Los estilos arquitecturales que hemos usado para dar forma a nuestro sistema, el tipo de aplicación a desarrollar y la infraestructura física determinarán en gran medida estas tecnologías. Por ejemplo, para hacer una aplicación de escritorio escogeremos WPF o Silverlight 3, o si nuestra aplicación expone su funcionalidad como servicios web, usaremos WCF. En resumen las preguntas que tenemos que responder son:

- ¿Qué tecnologías ayudan a implementar los estilos arquitecturales seleccionados?
- ¿Qué tecnologías ayudan a implementar el tipo de aplicación seleccionada?
- ¿Qué tecnologías ayudan a cumplir con los requisitos no funcionales especificados?

Lo más importante es ser capaz al terminar este punto de hacer un esquema de la arquitectura que refleje su estructura y las principales restricciones y decisiones de diseño tomadas. Esto permitirá establecer un marco para el sistema y discutir la solución propuesta.

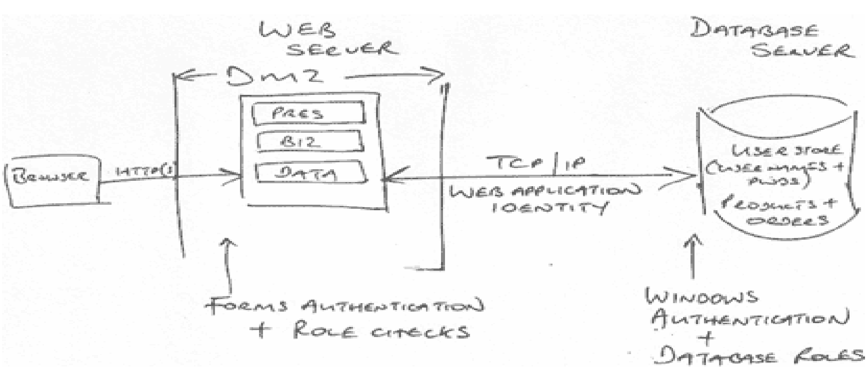


Figura 2.- Esquema de Arquitectura

4.- IDENTIFICAR LOS PRINCIPALES RIESGOS Y DEFINIR UNA SOLUCIÓN

El proceso de diseño de la arquitectura está dirigido por la funcionalidad, pero también por los riesgos a solventar. Cuanto antes minimicemos los riesgos, más probabilidades habrá de que tengamos éxito en nuestra arquitectura y al contrario.

La primera cuestión que surge es ¿Cómo identificar los riesgos de la arquitectura? Para responder a esta pregunta antes debemos tener claro qué requisitos no funcionales (o de calidad) tiene que tener nuestra aplicación. Esta información debería haber quedado definida tras la fase de inicio (*Inception*) y por lo tanto deberíamos contar con ella a la hora de realizar el diseño de la arquitectura.

Los requisitos no funcionales son aquellas propiedades que debe tener nuestra solución y que no son una funcionalidad, como por ejemplo: Alta disponibilidad, flexibilidad, interoperabilidad, mantenimiento, gestión operacional, rendimiento, fiabilidad, reusabilidad, escalabilidad, seguridad, robustez, capacidad de testeo y experiencia de usuario.

Es importante recalcar que normalmente nadie (un cliente normal) nos va a decir "la solución tiene que garantizar alta disponibilidad" sino que estos requisitos vendrán dados en el argot del cliente, por ejemplo "quiero que el sistema siga funcionando aunque falle algún componente", y es trabajo del arquitecto traducirlos o mejor dicho, enmarcarlos dentro de alguna de las categorías.

Ahora que tenemos claros qué requisitos no funcionales (y por tanto riesgos) debemos tratar, podemos proceder a definir una solución para mitigar cada uno de ellos. Los requisitos no funcionales tienen impacto en como nuestra arquitectura tratará determinados "puntos clave" como son: la autenticación y la autorización, el cacheo de datos y el mantenimiento del estado, las comunicaciones, la composición, la concurrencia y las transacciones, la gestión de la configuración, el acoplamiento y la cohesión, el acceso a datos, la gestión de excepciones, el registro de eventos y la instrumentalización del sistema, la experiencia de usuario, la validación de la información y el flujo de los procesos de negocio del sistema. Estos puntos clave si tendrán una funcionalidad asociada en algunos casos o determinarán como se realiza la implementación de un aspecto del sistema en otros.

Como hemos dicho, **los requisitos no funcionales son propiedades** de la solución **y no funcionalidad**, pero influyen directamente en **los puntos clave** de la arquitectura que **sí son funcionalidad** del sistema. Podemos decir que los requisitos no funcionales son la especificación de las propiedades de nuestro sistema y los puntos clave la implementación.

Por lo general un requisito no funcional tiene asociados varios puntos clave que influyen positiva o negativamente en su consecución. Por tanto, lo que haremos será analizar cada uno de los requisitos no funcionales en base a los "puntos clave" a los que afecta y tomaremos las decisiones apropiadas. Es importante entender que la relación entre requisitos no funcionales y puntos clave es M:M, esto significa que se producirán situaciones en las que un punto clave afecte a varios requisitos no funcionales. Cuando el punto clave sea beneficioso para la consecución de todos los requisitos no funcionales no habrá problema, pero cuando influya positivamente en uno

y negativamente en otro es donde se tendrán que tomar decisiones que establezcan un compromiso entre ambos requisitos.

Cada uno de estos atributos se ve más a fondo en el capítulo dedicado a los aspectos horizontales/transversales de la arquitectura.

Como ya hemos dicho, en esta fase del proyecto de diseño mitigamos los principales riesgos creando planes para solventarlos y planes de contingencia para el caso de que no puedan ser solventados. Para diseñar un plan para un requisito de calidad nos basaremos en los puntos clave a los que afecta dicho requisito. El plan de un requisito consistirá en una serie de decisiones sobre los puntos clave. Siempre que se pueda es mejor expresar estas decisiones de forma gráfica, por ejemplo en el caso de la seguridad indicando en el diagrama de arquitectura física el tipo de seguridad que se utiliza en cada zona o en el caso del rendimiento donde se realiza el cacheo de datos.

5.- CREAR ARQUITECTURAS CANDIDATAS

Una vez realizados los pasos anteriores, tendremos una arquitectura candidata que podremos evaluar. Si tenemos varias arquitecturas candidatas, realizaremos la evaluación de cada una de ellas e implementaremos la arquitectura mejor valorada. Cualquier arquitectura candidata debería responder a las siguientes preguntas:

- ¿Qué funcionalidad implementa?
- ¿Qué riesgos mitiga?
- ¿Cumple las restricciones impuestas por el cliente?
- ¿Qué cuestiones deja en el aire?

Si no es así, es que la arquitectura todavía no está bien definida o no hemos concretado alguno de los pasos anteriores.

Para valorar una arquitectura candidata nos fijaremos en la funcionalidad que implementa y en los riesgos que mitiga. Como en todo proceso de validación tenemos que establecer métricas que nos permitan definir criterios de satisfacibilidad. Para ello existen multitud de sistemas, pero en general tendremos 2 tipos de métricas: Cualitativas y cuantitativas.

Las métricas cuantitativas evaluarán un aspecto de nuestra arquitectura candidata y nos darán un índice que compararemos con el resto de arquitecturas candidatas y con un posible mínimo requerido.

Las métricas cualitativas evaluarán si la arquitectura candidata cumple o no con un determinado requisito funcional o de calidad de servicio de la solución. Generalmente serán evaluadas de forma binaria como un sí o un no.

Con estas dos métricas podremos crear métricas combinadas, como por ejemplo métricas cuantitativas que solo serán evaluadas tras pasar el sesgo de una métrica cualitativa.

Como ya hemos indicado existen multitud de sistemas para evaluar las arquitecturas software, pero todos ellos en mayor o menor medida se basan en este tipo de métricas. Los principales sistemas de evaluación de software son:

- *Software Architecture Analysis Method.*
- *Architecture Tradeoff Analysis Method.*
- *Active Design Review.*
- *Active Reviews of Intermediate Designs.*
- *Cost Benefit Analysis Method.*
- *Architecture Level Modifiability analysis.*
- *Family Architecture Assessment Method.*

Todas las decisiones sobre arquitectura deben plasmarse en una documentación que sea entendible por todos los integrantes del equipo de desarrollo así como por el resto de participantes del proyecto, incluidos los clientes. Hay muchas maneras de describir la arquitectura, como por ejemplo mediante ADL (*Architecture Description Language*), UML, Agile Modeling, IEEE 1471 o 4+1. Como dice el dicho popular, una imagen vale más que mil palabras, por ello nos decantamos por metodologías gráficas como 4+1. 4+1 describe una arquitectura software mediante 4 vistas distintas del sistema:

- **Vista lógica:** La vista lógica del sistema muestra la funcionalidad que el sistema proporciona a los usuarios finales. Emplea para ello diagramas de clases, de comunicación y de secuencia.

- **Vista del proceso**: La vista del proceso del sistema muestra cómo se comporta el sistema tiempo de ejecución, qué procesos hay activos y cómo se comunican. La vista del proceso resuelve cuestiones como la concurrencia, la escalabilidad, el rendimiento, y en general cualquier característica dinámica del sistema.

- **Vista física**: La vista física del sistema muestra cómo se distribuyen los distintos componentes software del sistema en los distintos nodos físicos de la infraestructura y cómo se comunican unos con otros. Emplea para ello los diagramas de despliegue.

- **Vista de desarrollo**: La vista lógica del sistema muestra el sistema desde el punto de vista de los programadores y se centra en el mantenimiento del software. Emplea para ello diagramas de componentes y de paquetes.

- **Escenarios:** La vista de escenarios completa la descripción de la arquitectura. Los escenarios describen secuencias de interacciones entre objetos y procesos y son usados para identificar los elementos arquitecturales y validar el diseño.

6.- ASPECTOS DE DOMAIN DRIVEN DESIGN



Figura 3.- Esquema de comunicación de Arquitectura

Hasta ahora hemos hablado del proceso de creación de la arquitectura, centrándonos en cómo elegir los casos de uso relevantes para la arquitectura, como decidir el tipo de aplicación que vamos a implementar y cómo afrontar los riesgos del proyecto dentro de la arquitectura. A continuación veremos aspectos claves que tenemos que tener en cuenta para conseguir una arquitectura que refleje nuestro dominio.

El objetivo de una arquitectura basada en Domain Driven Design es conseguir un modelo orientado a objetos que refleje el conocimiento de un dominio dado y que sea completamente independiente de cualquier concepto de comunicación, ya sea con elementos de infraestructura como de interfaz gráfica, etc. Buscamos construir un

modelo a través del cual podamos resolver problemas expresados como la colaboración de un conjunto de objetos. Por ello, debemos tener claro qué:

- Todo proyecto software con lógica compleja y un dominio complicado debe disponer de un modelo que represente los aspectos del dominio que nos permiten implementar los casos de uso.

- El foco de atención en nuestra arquitectura debe estar en el modelo del dominio y en la lógica del mismo, ya que este es una representación del conocimiento del problema.

- El modelo que construimos tiene que estar íntimamente ligado con la solución que entregamos, y por tanto tener en cuenta las consideraciones de implementación.

- Los modelos de dominio representan conocimiento acumulado, y dado que el conocimiento se adquiere de forma gradual e incremental, el proceso de creación de un modelo que represente profundamente los conceptos de un dominio debe ser iterativo.

6.1.- El lenguaje ubicuo

Uno de los principales motivos de fracaso de los proyectos software es la ruptura de la comunicación entre los expertos del dominio y los desarrolladores encargados de construir un sistema. La falta de un lenguaje común para comunicarse entre expertos del dominio y desarrolladores, así como entre los propios desarrolladores, genera problemas como la diferente interpretación de conceptos o la múltiple representación de un mismo concepto. Es esto lo que acaba derivando en implementaciones desconectadas del dominio con el que trabajan o en el que intentan resolver problemas. Las implementaciones desconectadas del dominio con el que trabajan presentan dos síntomas claramente observables:

- El sistema no resuelve correctamente un problema.

- El sistema no resuelve el problema adecuado.

Es vital tener claro, que cualquier modelo que construyamos debe estar profundamente representado en la implementación que hagamos del sistema, es decir, en lugar de disponer de un modelo de análisis y un modelo de implementación, debemos disponer de un único modelo, el modelo de dominio.

Cualquier modelo que construyamos debe representar de forma explícita los principales conceptos del dominio de conocimiento con el que trabaja nuestro sistema. Debemos fomentar la construcción de un lenguaje de uso común tanto entre expertos del dominio y desarrolladores, como entre los propios desarrolladores, que contenga

los principales conceptos del dominio de conocimiento con el que trabaja el sistema, y que sea el lenguaje usado para expresar cómo se resuelven los distintos problemas objetivo de nuestro sistema. Al utilizar un lenguaje común para comunicarnos, fomentamos la transferencia de conocimiento de los expertos del dominio a los desarrolladores, lo que permite que estos implementen un modelo de dominio mucho más profundo. Los buenos modelos se consiguen cuando los desarrolladores tienen un profundo conocimiento del dominio que están modelando, y este conocimiento solo se adquiere con el tiempo y a través de la comunicación con los expertos del dominio. Razón por la cual es imprescindible el uso de un lenguaje común.

6.2.- Prácticas que ayudan a conseguir un buen modelo de dominio.

El punto clave para un proyecto exitoso es la transferencia efectiva de conocimiento desde los expertos del dominio a los desarrolladores del sistema. Para facilitar esta transferencia de conocimiento podemos emplear varias técnicas de desarrollo conocidas.

6.2.1.- Behavior Driven Development (BDD)

BDD es una práctica aplicable dentro de cualquier metodología que consiste en la descripción de los requisitos como un conjunto de pruebas ejecutables de forma automática. BDD ayuda a la transferencia del conocimiento al provocar que los principales conceptos del dominio presentes en los requisitos, pasen directamente al código, destacando su importancia dentro del dominio y creando un contexto o base para la construcción del modelo.

6.2.2.- Test Driven Development (TDD)

TDD es una práctica aplicable dentro de cualquier metodología que consiste en desarrollar un conjunto de pruebas que sirven como especificación y justificación de la necesidad de crear un componente para implementar una determinada funcionalidad. Estas pruebas permiten definir la forma del propio componente e indagan en las relaciones de este con otros componentes del dominio, lo que fomenta el desarrollo del modelo.

CAPÍTULO

3

Arquitectura Marco N-Capas

1.- ARQUITECTURA DE APLICACIONES EN N-CAPAS

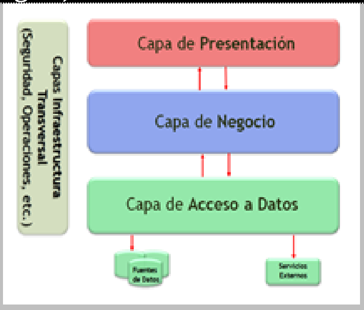
1.1.- Capas vs. Niveles (Layers vs. Tiers)

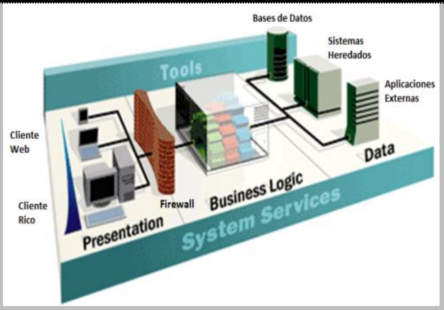
Es importante distinguir los conceptos de "Capas" (*Layers*) y "Niveles" (*Tiers*), pues es bastante común que se confundan y o se denominen de forma incorrecta.

Las Capas (*Layers*) se ocupan de la división lógica de componentes y funcionalidad, y no tienen en cuenta la localización física de componentes en diferentes servidores o en diferentes lugares. Por el contrario, los Niveles (*Tiers*) se ocupan de la distribución física de componentes y funcionalidad en servidores separados, teniendo en cuenta topología de redes y localizaciones remotas. Aunque tanto las Capas (*Layers*) como los Niveles (*Tiers*) usan conjuntos similares de nombres (presentación, servicios, negocio y datos), es importante no confundirlos y recordar que solo los Niveles (*Tiers*) implican una separación física. Se suele utilizar el término "*Tier*" refiriéndonos a patrones de distribución física como "2 *Tier*", "3-*Tier*" y "N-*Tier*".

A continuación mostramos un esquema **3-Tier** y un esquema **N-Layer** donde se pueden observar las diferencias comentadas (lógica vs. Situación física):

Tabla 1.- N-Tier vs. N-Layer

<table>
<tr><th>Arquitectura tradicional N-Capas (Lógica)</th><th>Arquitectura 3-Tier (Física)</th></tr>
<tr><td>

</td><td>

</td></tr>
<tr><td>Figura 1.- Arquitectura tradicional N-Layer (Lógica)</td><td>Figura 2.- Arquitectura 3-Tier (Física)</td></tr>
</table>

Por último, destacar que todas las aplicaciones con cierta complejidad, deberían implementar una arquitectura lógica de tipo N-Capas, pues proporciona una estructuración lógica correcta; sin embargo, no todas las aplicaciones tienen por qué implementarse en modo *N-Tier*, puesto que hay aplicaciones que no requieren de una separación física de sus niveles (*Tiers*), como pueden ser muchas aplicaciones web.

1.2.- Capas

Contexto

Se quiere diseñar una aplicación empresarial compleja compuesta por un número considerable de componentes de diferentes niveles de abstracción.

Problema

Cómo estructurar una aplicación para soportar requerimientos complejos operacionales y disponer de una buena mantenibilidad, reusabilidad, escalabilidad, robustez y seguridad.

Aspectos relacionados

Al estructurar una aplicación, se deben reconciliar las siguientes 'fuerzas' dentro del contexto del entorno de la aplicación:

- Localizar los cambios de un tipo en una parte de la solución minimiza el impacto en otras partes, reduce el trabajo requerido en arreglar defectos, facilita el mantenimiento de la aplicación y mejora la flexibilidad general de la aplicación.

- Separación de responsabilidades entre componentes (por ejemplo, separar la interfaz de usuario de la lógica de negocio, y la lógica de negocio del acceso a la base de datos) aumenta la flexibilidad, la mantenibilidad y la escalabilidad.

- Ciertos componentes deben ser reutilizables entre diferentes módulos de una aplicación o incluso entre diferentes aplicaciones.

- Equipos diferentes deben poder trabajar en partes de la solución con mínimas dependencias entre los diferentes equipos de desarrollo y para ello, deben desarrollar contra interfaces bien definidas.

- Los componentes individuales deben ser cohesivos

- Los componentes no relacionados directamente deben estar débilmente acoplados

- Los diferentes componentes de una solución deben poder ser desplegados de una forma independiente, e incluso mantenidos y actualizados en diferentes momentos.

- Para asegurar estabilidad y calidad, cada capa debe contener sus propias pruebas unitarias.

Las capas son agrupaciones horizontales lógicas de componentes de software que forman la aplicación o el servicio. Nos ayudan a diferenciar entre los diferentes tipos de tareas a ser realizadas por los componentes, ofreciendo un diseño que maximiza la reutilización y, especialmente, la mantenibilidad. En definitiva, se trata de aplicar el principio de '*Separación de Responsabilidades'* (SoC - *Separation of Concerns principle*) dentro de una Arquitectura.

Cada capa lógica de primer nivel puede tener un número concreto de componentes agrupados en sub-capas. Dichas sub-capas realizan a su vez un tipo específico de tareas. Al identificar tipos genéricos de componentes que existen en la mayoría de las soluciones, podemos construir un patrón o mapa de una aplicación o servicio y usar dicho mapa como modelo de nuestro diseño.

El dividir una aplicación en capas separadas que desempeñan diferentes roles y funcionalidades, nos ayuda a mejorar el mantenimiento del código; nos permite también diferentes tipos de despliegue y, sobre todo, nos proporciona una clara

delimitación y situación de dónde debe estar cada tipo de componente funcional e incluso cada tipo de tecnología.

Diseño básico de capas

Se deben separar los componentes de la solución en capas. Los componentes de cada capa deben ser cohesivos y tener aproximadamente el mismo nivel de abstracción. Cada capa de primer nivel debe de estar débilmente acoplada con el resto de capas de primer nivel. El proceso es como sigue:

Comenzar en el nivel más bajo de abstracción, por ejemplo 'Capa 1'. Esta capa es la base del sistema. Se continúa esta escalera abstracta con otras capas (Capa J, Capa J-1) hasta el último nivel (Capa-N):

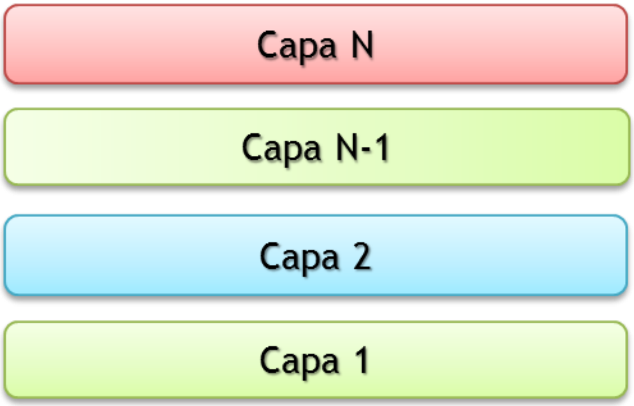


Figura 3.- Diseño básico de capas

La clave de una aplicación en N-Capas está en la gestión de dependencias. En una arquitectura N-Capas tradicional, los componentes de una capa pueden interactuar solo con componentes de la misma capa o bien con otros componentes de capas inferiores. Esto ayuda a reducir las dependencias entre componentes de diferentes niveles. Normalmente hay **dos aproximaciones al diseño en capas: Estricto y Laxo**.

Un **'diseño en Capas estricto'** limita a los componentes de una capa a comunicarse solo con los componentes de su misma capa o con la capa inmediatamente inferior. Por ejemplo, en la figura anterior, si utilizamos el sistema estricto, la capa J solo podría interactuar con los componentes de la capa J-1, la capa J-1 solo con los componentes de la capa J-2, y así sucesivamente.

Un **'diseño en Capas laxo'** permite que los componentes de una capa interactúen con cualquier otra capa de nivel inferior. Por ejemplo, en la figura anterior, si utilizamos esta aproximación, la capa J podría interactuar con la capa J-1, J-2 y J-3.

El uso de la aproximación laxa puede mejorar el rendimiento porque el sistema no tiene que realizar redundancia de llamadas de unas capas a otras. Por el contrario, el uso de la aproximación laxa no proporciona el mismo nivel de aislamiento entre las

diferentes capas y hace más difícil el sustituir una capa de más bajo nivel sin afectar a muchas más capas de nivel superior (y no solo a una).

En soluciones grandes que involucran a muchos componentes de software, es habitual tener un gran número de componentes en el mismo nivel de abstracción (capas) pero que sin embargo no son cohesivos. En esos casos, cada capa debería descomponerse en dos o más subsistemas cohesivos, llamados también Módulos (parte de un módulo vertical en cada capa horizontal). El concepto de módulo lo explicamos en más detalle posteriormente dentro de la Arquitectura marco propuesta, en este mismo capítulo.

El siguiente diagrama UML representa capas compuestas a su vez por múltiples subsistemas:

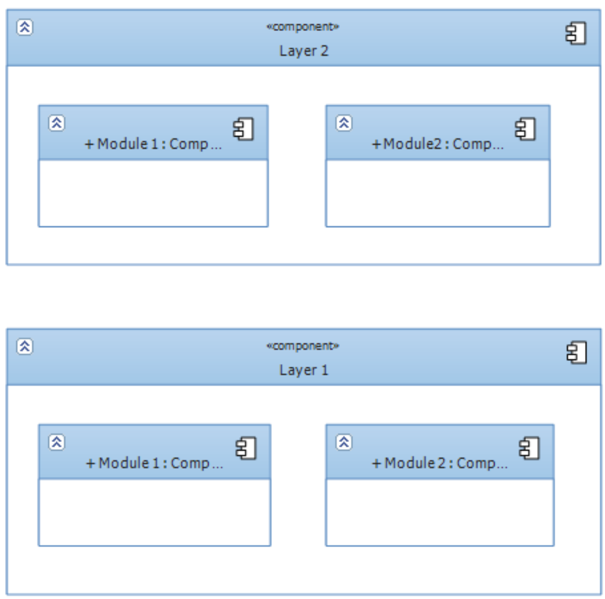


Figura 4.- Múltiples subsistemas en cada capa

Consideraciones relativas a Pruebas

Una aplicación en N-Capas mejora considerablemente la capacidad de implementar pruebas de una forma apropiada:

- Debido a que cada capa interactúa con otras capas solo mediante interfaces bien definidos, es fácil añadir implementaciones alternativas a cada capa (*Mock* y *Stubs*). Esto permite realizar pruebas unitarias de una capa incluso cuando las capas de las que depende no están finalizadas o, incluso, porque se quiera poder ejecutar mucho más rápido un conjunto muy grande de pruebas unitarias que al acceder a las capas de las que depende se ejecutan mucho más

lentamente. Esta capacidad se ve muy mejorada si se hace uso de clases base (Patrón '*Layered Supertype*') e interfaces (Patrón '*Abstract Interface*'), porque limitan aún más las dependencias entre las capas., Es especialmente importante el uso de interfaces pues nos dará la posibilidad de utilizar técnicas más avanzadas de desacoplamiento, que exponemos más adelante en esta guía.

- Es más fácil realizar pruebas sobre componentes individuales porque las dependencias entre ellos están limitadas de forma que los componentes de capas de alto nivel solo pueden interaccionar con los de niveles inferiores. Esto ayuda a aislar componentes individuales para poder probarlos adecuadamente y nos facilita el poder cambiar unos componentes de capas inferiores por otros diferentes con un impacto muy pequeño en la aplicación (siempre y cuando cumplan los mismos interfaces).

Beneficios de uso de Capas

- El mantenimiento de mejoras en una solución será mucho más fácil porque las funciones están localizadas. Además las capas deben estar débilmente acopladas entre ellas y con alta cohesión internamente, lo cual posibilita variar de una forma sencilla diferentes implementaciones/combinaciones de capas.

- Otras soluciones deberían poder reutilizar funcionalidad expuesta por las diferentes capas, especialmente si se han diseñado para ello.

- Los desarrollos distribuidos son mucho más sencillos de implementar si el trabajo se ha distribuido previamente en diferentes capas lógicas.

- La distribución de capas (*layers*) en diferentes niveles físicos (*tiers*) puede, en algunos casos, mejorar la escalabilidad. Aunque este punto hay que evaluarlo con cuidado, pues puede impactar negativamente en el rendimiento.

Referencias

Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerland, Peter; and Stal, Michael. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley & Sons, 1996.

Fowler, Martin. Patterns ofApplication Architecture. Addison-Wesley, 2003.

Gamma, Eric; Helm, Richard; Johnson, Ralph; and Vlissides, John. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

1.3.- Principios Base de Diseño a seguir

A la hora de diseñar un sistema, es importante tener presente una serie de principios de diseño fundamentales que nos ayudarán a crear una arquitectura que se ajuste a prácticas demostradas, que minimicen los costes de mantenimiento y maximicen la usabilidad y la extensibilidad. Estos principios clave seleccionados y muy reconocidos por la industria del *software*, son:

1.3.1.- Principios de Diseño 'SOLID'

El acrónimo SOLID deriva de las siguientes frases/principios en inglés:

Principios '*SOLID*' en Diseño

- ❑ ***S****ingle Responsability Principle*
- ❑ ***O****pen Close Principle*
- ❑ ***L****iskov Substitution Principle*
- ❑ ***I****nterface Segregation Principle*
- ❑ ***D****ependency Inversion Principle*

De una forma resumida, los principios de diseño SOLID son los siguientes:

- **Principio de Única Responsabilidad ('*Single Responsability Principle*')**: *Una clase debe tener una única responsabilidad o característica*. Dicho de otra manera, una clase debe de tener una única razón por la que está justificado realizar cambios sobre su código fuente. Una consecuencia de este principio es que, de forma general, las clases deberían tener pocas dependencias con otras clases/tipos.

- **Principio Abierto Cerrado (‘*Open Closed Principle*’):** *Una clase debe estar abierta para la extensión y cerrada para la modificación.* Es decir, el comportamiento de una clase debe poder ser extendido sin necesidad de realizar modificaciones sobre el código de la misma.

- **Principio de Sustitución de Liskov (‘*Liskov Subtitution Principle*’):** *Los subtipos deben poder ser sustituibles por sus tipos base (interfaz o clase base).* Este hecho se deriva de que el comportamiento de un programa que trabaja con abstracciones (interfaces o clases base) no debe cambiar porque se sustituya una implementación concreta por otra. Los programas deben hacer referencia a las abstracciones, y no a las implementaciones. Veremos posteriormente que este principio va a estar muy relacionado con la *Inyección de Dependencias* y la sustitución de unas clases por otras siempre que cumplan el mismo interfaz.

- **Principio de Segregación de Interfaces (‘*Interface Segregation Principle’*):** *Los implementadores de Interfaces de clases no deben estar obligados a implementar métodos que no se usan*. Es decir, los interfaces de clases deben ser específicos dependiendo de quién los consume y por lo tanto, tienen que estar granularizados/segregados en diferentes interfaces no debiendo crear nunca grandes interfaces. Las clases deben exponer interfaces separados para diferentes clientes/consumidores que difieren en los requerimientos de interfaces.

- **Principio de Inversión de Dependencias (‘*Dependency Inversion Principle*’):** *Las abstracciones no deben depender de los detalles – Los detalles deben depender de las abstracciones*. Las dependencias directas entre clases deben ser reemplazadas por abstracciones (interfaces) para permitir diseños top-down sin requerir primero el diseño de los niveles inferiores.

1.3.2.- Otros Principios clave de Diseño

- **El diseño de componentes debe ser altamente cohesivo:** no sobrecargar los componentes añadiendo funcionalidad mezclada o no relacionada. Por ejemplo, evitar mezclar lógica de acceso a datos con lógica de negocio perteneciente al Modelo del Dominio. Cuando la funcionalidad es cohesiva, entonces podemos crear ensamblados/*assemblies* que contengan más de un componente y situar los componentes en las capas apropiadas de la aplicación. Este principio está por lo tanto muy relacionado con el patrón ‘N-Capas’ y con el principio de ‘*Single Responsability Principle*’.

- **Mantener el código transversal abstraído de la lógica específica de la aplicación**: el código transversal se refiere a código de aspectos horizontales, cosas como la seguridad, gestión de operaciones, logging, instrumentalización, etc. La mezcla de este tipo de código con la implementación específica de la aplicación puede dar lugar a diseños que sean en el futuro muy difíciles de extender y mantener. Relacionado con este principio está AOP (*Aspect Oriented Programming*).

- **Separación de Preocupaciones/Responsabilidades (‘*Separation of Concerns*’):** dividir la aplicación en distintas partes minimizando las funcionalidades superpuestas ente dichas partes. El factor fundamental es minimizar los puntos de interacción para conseguir una alta cohesión y un bajo acoplamiento. Sin embargo, separar la funcionalidad en las fronteras equivocadas, puede resultar en un alto grado de acoplamiento y complejidad entre las características del sistema.

- **No repetirse (DRY)**: se debe especificar ‘la intención’ en un único sitio en el sistema. Por ejemplo, en términos del diseño de una aplicación, una funcionalidad específica se debe implementar en un único componente; esta misma funcionalidad no debe estar implementada en otros componentes.

- **Minimizar el diseño de arriba abajo** (*Upfront design*): diseñar solamente lo que es necesario, no realizar ‘sobre-ingenierías’ y evitar el efecto **YAGNI** (En inglés-*slang*: ***Y****ou* ***A****in’t* ***G****onna* ***N****eed* ***I****t*).

1.4.- Orientación a tendencias de Arquitectura DDD (*Domain Driven Design*)

El objetivo de esta arquitectura marco es proporcionar una base consolidada y guías de arquitectura para un tipo concreto de aplicaciones: **‘Aplicaciones empresariales complejas’**. Este tipo de aplicaciones se caracterizan por tener una vida relativamente larga y un volumen de cambios evolutivos considerable. Por lo tanto, en estas aplicaciones es muy importante todo lo relativo al mantenimiento de la aplicación, la facilidad de actualización, o la sustitución de tecnologías y *frameworks/ORMs* (*Object-relational mapping*) por otras versiones más modernas o incluso por otros diferentes, etc. El objetivo es que todo esto se pueda realizar con el menor impacto posible sobre el resto de la aplicación. En definitiva, que los cambios de tecnologías de infraestructura de una aplicación no afecten a capas de alto nivel de la aplicación, especialmente, que afecten lo mínimo posible a la capa del ‘Dominio de la aplicación’.

En las aplicaciones complejas, el comportamiento de las reglas de negocio (lógica del Dominio) está sujeto a muchos cambios y es muy importante poder modificar, construir y realizar pruebas sobre dichas capas de lógica del dominio de una forma fácil e independiente. Debido a esto, un objetivo importante es tener el mínimo acoplamiento entre el Modelo del Dominio (lógica y reglas de negocio) y el resto de capas del sistema (Capas de presentación, Capas de Infraestructura, persistencia de datos, etc.).

Debido a las premisas anteriores, las tendencias de arquitectura de aplicaciones que están más orientadas a conseguir este desacoplamiento entre capas, especialmente la independencia y foco preponderante sobre la capa del Modelo de Domino, son precisamente las Arquitecturas N-Capas Orientadas al Dominio, como parte de DDD (*Domain Driven Design*).

DDD (Domain Driven Design) es, sin embargo, mucho más que simplemente una Arquitectura propuesta; es también una forma de afrontar los proyectos, una forma de trabajar por parte del equipo de desarrollo, la importancia de identificar un 'Lenguaje Ubicuo' proyectado a partir del conocimiento de los expertos en el dominio (expertos en el negocio), etc. Sin embargo, todo esto queda fuera de la presente guía puesto que se quiere limitar a una Arquitectura lógica y tecnológica, no a la forma de afrontar un proyecto de desarrollo o forma de trabajar de un equipo de desarrollo. Todo esto puede consultarse en libros e información relacionada con DDD.

Razones por las que no se debe orientar a Arquitecturas N-Capas Orientadas al Dominio

Debido a las premisas anteriores, se desprende que si la aplicación a realizar es relativamente sencilla y, sobre todo, si las reglas de negocio a automatizar en la aplicación cambiarán muy poco y no se prevén necesidades de cambios de tecnología de infraestructura durante la vida de dicha aplicación, entonces, probablemente la solución no debería seguir el tipo de arquitectura presentado en esta guía, y más bien se debería seleccionar un tipo de desarrollo/tecnología RAD (*Rapid Application Development*), como puede ser '*WCF RIA Services'*. Es decir, tecnologías de rápida implementación a ser utilizadas para construir aplicaciones sencillas donde el desacoplamiento entre todos sus componentes y capas no es especialmente relevante, pero sí lo es facilidad y productividad en el desarrollo y el '*time to market'*. De forma generalista se suele decir que son aplicaciones centradas en datos (*Data Driven*) y no tanto en un modelo de dominio (*Domain Driven Design*).

Razones por las que se si se debe orientar a Arquitectura N-Capas Orientada al Dominio

Es realmente volver hacer hincapié sobre lo mismo, pero es muy importante dejar este aspecto claro.

Así pues, las razones por las que es importante hacer uso de una 'Arquitectura N-Capas Orientada al Dominio' es especialmente en los casos donde el comportamiento

del negocio a automatizar (lógica del dominio) está sujeto a muchos cambios y evoluciones. En este caso específico, disponer de un 'Modelo de Dominio' disminuirá el coste total de dichos cambios, y a medio plazo el TCO (Coste Total de la Propiedad) será mucho menor que si la aplicación hubiera sido desarrollada de una forma más acoplada, porque los cambios no tendrán tanto impacto. En definitiva, el tener todo el comportamiento del negocio que puede estar cambiando encapsulado en una única área de nuestro software, disminuye drásticamente la cantidad de tiempo que se necesita para realizar un cambio. Porque este cambio se realizará en un solo sitio y podrá ser convenientemente probado de forma aislada, aunque esto por supuesto dependerá de cómo se haya desarrollado. El poder aislar tanto como sea posible dicho código del Modelo del Dominio disminuye las posibilidades de tener que realizar cambios en otras áreas de la aplicación (lo cual siempre puede afectar con nuevos problemas, regresiones, etc.). Esto es de vital importancia si se desea reducir y mejorar los ciclos de estabilización y puesta en producción de las soluciones.

Escenarios donde utilizar el Modelo de Dominio

Las reglas de negocio que indican cuándo se permiten ciertas acciones son precisamente buenas candidatas a ser implementadas en el modelo de dominio.

Por ejemplo, en un sistema comercial, una regla que especifica que un cliente no puede tener pendiente de pago más de 2.000€, probablemente debería pertenecer al modelo de dominio. Hay que tener en cuenta que reglas como la anterior involucran a diferentes entidades y tienen que evaluarse en diferentes casos de uso.

Así pues, en un modelo de dominio tendremos muchas de estas reglas de negocio, incluyendo casos donde unas reglas sustituyen a otras. Por ejemplo, sobre la regla anterior, si el cliente es una cuenta estratégica o con un volumen de negocio muy grande dicha cantidad podría ser muy superior, etc.

En definitiva, la importancia que tengan en una aplicación las reglas de negocio y los casos de uso es precisamente la razón por la que orientar la arquitectura hacia el Dominio y no simplemente definir entidades, relaciones entre ellas y una aplicación orientada a datos.

Finalmente, para persistir la información y convertir colecciones de objetos en memoria (grafos de objetos/entidades) a una base de datos relacional, podemos hacer uso de alguna tecnología de persistencia de datos de tipo ORM (*Object-Relational Mapping*), como *NHibernate* o *Entity Framework*. Sin embargo, es muy importante que queden muy diferenciadas y separadas estas tecnologías concretas de persistencia de datos (tecnologías de infraestructura) del comportamiento de negocio de la aplicación, que es responsabilidad del Modelo del Dominio. Para esto, se necesita una arquitectura en Capas (*N-Layer*) que esté integrada de una forma desacoplada, como veremos posteriormente.

1.5.- DDDD (*Distributed Domain Driven Design*)

¿Cuatro 'D'?. Bueno, sí, está claro que DDDD es una evolución/extensión de DDD donde se añaden aspectos de sistemas distribuidos. Eric Evans, en su libro de DDD obvia casi por completo los sistemas y tecnologías distribuidas, (Servicios Web, etc.) porque se centra mayoritariamente en el Dominio. Sin embargo, los sistemas distribuidos y Servicios remotos son algo que necesitamos en la mayoría de los escenarios.

Realmente, la presente propuesta de Arquitectura N-Layer, está basada en **DDDD**, porque tenemos en cuenta desde el principio a la capa de Servicios Distribuidos, e incluso lo mapeamos luego a implementación con tecnología Microsoft.

En definitiva, esta cuarta D añadida a DDD nos acerca a escenarios distribuidos, gran escalabilidad e incluso escenarios que normalmente se acercarán a '*Cloud-Computing*' pos su afinidad.

2.- ARQUITECTURA MARCO N-CAPAS CON ORIENTACIÓN AL DOMINIO

Queremos recalcar que hablamos de arquitectura con 'Orientación al Dominio', no hablamos de todo lo que cubre DDD (*Domain Driven Design*). Para hablar de DDD deberíamos centrarnos realmente no solo en la arquitectura (objetivo de esta guía), sino más bien en el proceso de diseño, en la forma de trabajar de los equipos de desarrollo, el 'lenguaje ubicuo', etc. Esos aspectos de DDD los tocaremos en la presente guía, pero de forma leve. **El objetivo de esta guía es centrarnos exclusivamente en una *Arquitectura N-Layer* que encaje con DDD, y como "mapearlo" posteriormente a las tecnologías Microsoft. No pretendemos exponer y explicar DDD, para esto último ya existen magníficos libros al respecto.**

Esta sección define de forma global la arquitectura marco en N-Capas así como ciertos patrones y técnicas a tener en cuenta para la integración de dichas capas.

2.1.- Capas de Presentación, Aplicación, Dominio e Infraestructura

En el nivel más alto y abstracto, la vista de arquitectura lógica de un sistema puede considerarse como un conjunto de servicios relacionados agrupados en diversas capas, similar al siguiente esquema (siguiendo las tendencias de Arquitectura DDD):

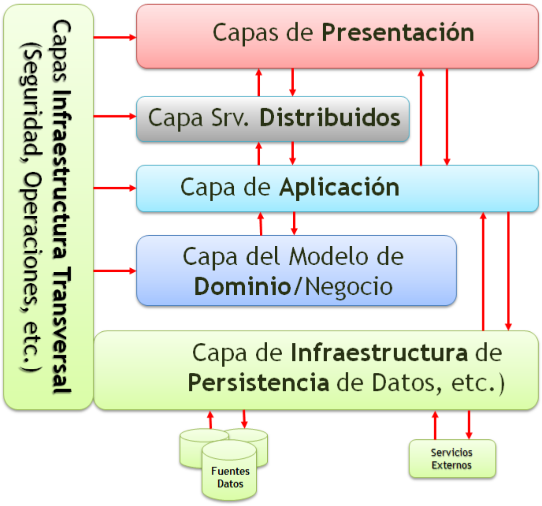


Figura 5.- Vista de arquitectura lógica simplificada de un sistema N-Capas DDD

En Arquitecturas 'Orientadas al Dominio' es crucial la clara delimitación y separación de la capa del Dominio del resto de capas. Es realmente un pre-requisito para DDD. "*Todo debe girar alrededor del Dominio*".

Así pues, se debe particionar una aplicación compleja en capas. Desarrollar un diseño dentro de cada capa que sea cohesivo, pero delimitando claramente las diferentes capas entre ellas, aplicando patrones estándar de Arquitectura para que dichas dependencias sean en muchas ocasiones basadas en abstracciones y no referenciando una capa directamente a la otra. Concentrar todo el código relacionado con el modelo del dominio en una capa y aislarlo del resto de código de otras capas (Presentación, Aplicación, Infraestructura y Persistencia, etc.). Los objetos del Dominio, al estar libres de tener que mostrarse ellos mismos, persistirse/guardarse, gestionar tareas de aplicación, etc. pueden entonces centrarse exclusivamente en expresar el modelo de dominio. Esto permite que un modelo de dominio pueda evolucionar y llegar a ser lo suficientemente rico y claro para representar el conocimiento de negocio esencial y ponerlo realmente en ejecución dentro de la aplicación.

El separar la capa de dominio del resto de capas permite un diseño mucho más limpio de cada capa. Las capas aisladas son mucho menos costosas de mantener porque tienden a evolucionar a diferentes ritmos y responder a diferentes necesidades. Por ejemplo, las capas de infraestructura evolucionarán cuando evolucionen las tecnologías sobre las que están basadas. Por el contrario, la capa del Dominio evolucionará solo cuando se quieran realizar cambios en la lógica de negocio del Dominio concreto.

Adicionalmente, *la separación de capas ayuda en el despliegue de un sistema distribuido, permitiendo que diferentes capas sean situadas de forma flexible en diferentes servidores o clientes, de manera que se minimice el exceso de comunicación y se mejore el rendimiento (Cita de M. Fowler).*

La integración y desacoplamiento entre las diferentes capas de alto nivel es algo fundamental. Cada capa de la aplicación contendrá una serie de componentes que implementan la funcionalidad de dicha capa. Estos componentes deben ser cohesivos internamente (dentro de la misma capa de primer nivel), pero algunas capas (como las capas de Infraestructura/Tecnología) deben estar débilmente acopladas con el resto de capas para poder potenciar las pruebas unitarias, *mocking*, la reutilización y finalmente que impacte menos al mantenimiento. Este desacoplamiento entre las capas principales se explica en más detalle posteriormente, tanto su diseño como su implementación.

2.2.- Arquitectura marco N-Capas con Orientación al Dominio

El objetivo de esta arquitectura marco es estructurar de una forma limpia y clara la complejidad de una aplicación empresarial basada en las diferentes capas de la arquitectura, siguiendo el patrón *N-Layered* y las tendencias de arquitecturas en DDD. El patrón *N-Layered* distingue diferentes capas y sub-capas internas en una aplicación, delimitando la situación de los diferentes componentes por su tipología.

Por supuesto, esta arquitectura concreta N-Layer es personalizable según las necesidades de cada proyecto y/o preferencias de Arquitectura. Simplemente proponemos una Arquitectura marco a seguir que sirva como punto base a ser modificada o adaptada por arquitectos según sus necesidades y requisitos.

En concreto, las capas y sub-capas propuestas para aplicaciones '*N-Layered con Orientación al Dominio*' son:

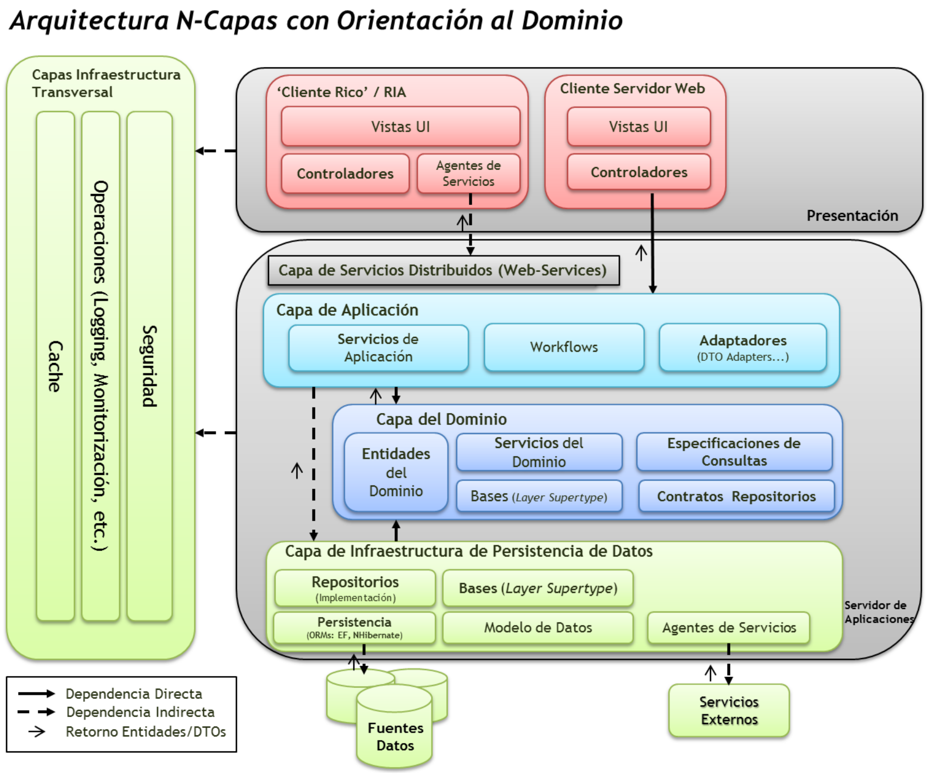


Figura 6.- Arquitectura N-Capas con Orientación al Dominio

- **Capa de Presentación**
    - Subcapas de Componentes Visuales (Vistas)
    - Subcapas de Proceso de Interfaz de Usuario (Controladores y similares)
- **Capa de Servicios Distribuidos (Servicios-Web)**
    - Servicios-Web publicando las Capas de Aplicación y Dominio
- **Capa de Aplicación**
    - Servicios de Aplicación (Tareas y coordinadores de casos de uso)
    - Adaptadores (Conversores de formatos, etc.)
    - Subcapa de *Workflows* (Opcional)
    - Clases base de Capa Aplicación (Patrón Layer-Supertype)

- **Capa del Modelo de Dominio**
  - Entidades del Dominio
  - Servicios del Dominio
  - Especificaciones de Consultas (Opcional)
  - Contratos/Interfaces de *Repositorios*
  - Clases base del Dominio (Patrón *Layer-Supertype*)
- **Capa de Infraestructura de Acceso a Datos**
  - Implementación de Repositorios'
  - Modelo lógico de Datos
  - Clases Base (Patrón *Layer-Supertype*)
  - Infraestructura tecnología ORM
  - Agentes de Servicios externos
- **Componentes/Aspectos Horizontales de la Arquitectura**
  - Aspectos horizontales de Seguridad, Gestión de operaciones, Monitorización, Correo Electrónico automatizado, etc.

Todas estas capas se explican en el presente capítulo de forma breve y posteriormente dedicamos un capítulo a cada una de ellas; sin embargo, antes de ello, es interesante conocer desde un punto de vista de alto nivel cómo es la interacción entre dichas capas y por qué las hemos dividido así.

Una de las fuentes y precursores principales de DDD, es *Eric Evans*, el cual en su libro "*Domain Driven Design - Tackling Complexity in the Heart of Software*" expone y explica el siguiente diagrama de alto nivel con su propuesta de Arquitectura N-Layer:

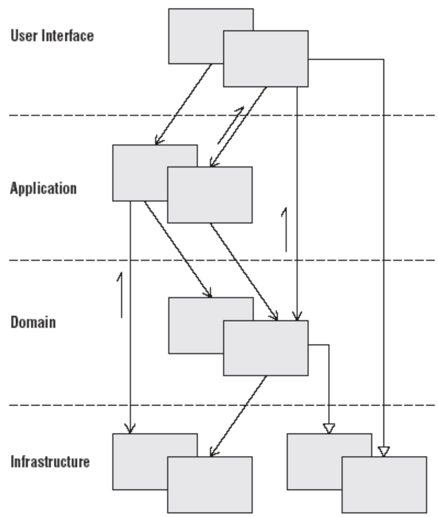


Figura 7.- Diagrama de Arquitectura N-Layer DDD

Es importante resaltar que en algunos casos el acceso a las otras capas es directo. Es decir, no tiene por qué haber un camino único obligatorio pasando de una capa a otra, aunque dependerá de los casos. Para que queden claros dichos casos a continuación mostramos el anterior diagrama de Eric-Evans, pero modificado y un poco más detallado, de forma que se relaciona con las sub-capas y elementos de más bajo nivel que proponemos en nuestra Arquitectura:

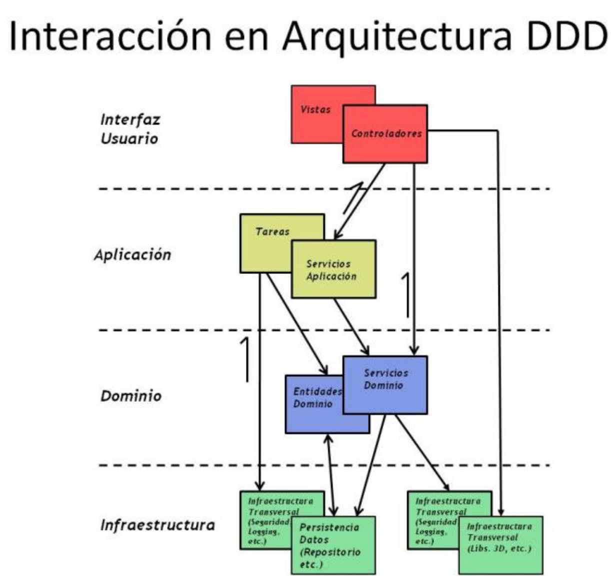


Figura 8.- Interacción en Arquitectura DDD

Primeramente, podemos observar que la **Capa de Infraestructura** que presenta una arquitectura con tendencia DDD, es algo muy amplio y para muchos contextos muy diferentes (Contextos de Servidor y de Cliente). La Capa de infraestructura contendrá todo lo ligado a tecnología/infraestructura. Ahí se incluyen conceptos fundamentales como Persistencia de Datos (Repositorios, etc.), pasando por aspectos transversales como Seguridad, *Logging*, Operaciones, etc. e incluso podría llegar a incluirse librerías específicas de capacidades gráficas para UX (librerías 3D, librerías de controles específicos para una tecnología concreta de presentación, etc.). Debido a estas grandes diferencias de contexto y a la importancia del acceso a datos, en nuestra arquitectura propuesta hemos separado explícitamente la Capa de Infraestructura de '**Persistencia de Datos'** del resto de capas de '**Infraestructura Transversal'**, que pueden ser utilizadas de forma horizontal/transversal por cualquier capa.

El otro aspecto interesante que adelantábamos anteriormente, es el hecho de que el acceso a algunas capas no es con un único camino ordenado por diferentes capas. Concretamente podremos acceder directamente a las capas de Aplicación, de Dominio y de Infraestructura Transversal siempre que lo necesitemos. Por ejemplo, podríamos acceder directamente desde una Capa de Presentación Web (no necesita interfaces remotos de tipo Servicio-Web) a las capas inferiores que necesitemos (Aplicación, Dominio, y algunos aspectos de Infraestructura Transversal). Sin embargo, para llegar a la '*Capa de Persistencia de Datos'* y sus objetos *Repositorios* (puede recordar en algunos aspectos a la *Capa de Acceso a Datos (DAL)* tradicional, pero no es lo mismo), es recomendable que siempre se acceda a través de los objetos de coordinación (Servicios) de la Capa de Aplicación, puesto que es la parte que los orquesta.

Queremos resaltar que la implementación y uso de todas estas capas debe ser algo flexible. Relativo al diagrama, probablemente deberían existir más combinaciones de flechas (accesos). Y sobre todo, no tiene por qué ser utilizado de forma exactamente igual en todas las aplicaciones.

A continuación, en este capítulo describimos brevemente cada una de las capas y subcapas mencionadas. También presentamos algunos conceptos globales de cómo definir y trabajar con dichas capas (desacoplamiento entre algunas capas, despliegue en diferentes niveles físicos, etc.).

Posteriormente, en los próximos capítulos se procederá a definir y explicar en detalle cada una de dichas capas de primer nivel (Un capítulo por cada capa de primer nivel).

Capa de Presentación

Esta capa es responsable de mostrar información al usuario e interpretar sus acciones.

Los componentes de las capas de presentación implementan la funcionalidad requerida para que los usuarios interactúen con la aplicación. Normalmente es recomendable subdividir dichos componentes en varias sub-capas aplicando patrones de tipo MVC, MVP o M-V-VM:

- **Subcapa de Componentes Visuales (Vistas):** Estos componentes proporcionan el mecanismo base para que el usuario utilice la aplicación. Son componentes que formatean datos en cuanto a tipos de letras y controles visuales, y también reciben datos proporcionados por el usuario.

- **Subcapa de Controladores:** Para ayudar a sincronizar y orquestar las interacciones del usuario, puede ser útil conducir el proceso utilizando componentes separados de los componentes propiamente gráficos. Esto impide que el flujo de proceso y lógica de gestión de estados esté programada dentro de los propios controles y formularios visuales y permite reutilizar dicha lógica y patrones desde otros interfaces o 'vistas'. También es muy útil para poder realizar pruebas unitarias de la lógica de presentación. Estos '*Controllers'* son típicos de los patrones MVC y derivados.

Capa de Servicios Distribuidos (Servicios Web) –Opcional-

Cuando una aplicación actúa como proveedor de servicios para otras aplicaciones remotas, o incluso si la capa de presentación esta también localizada físicamente en localizaciones remotas (aplicaciones *Rich-Client*, RIA, OBA, etc.), normalmente se publica la lógica de negocio (capas de negocio internas) mediante una capa de servicios. Esta capa de servicios (habitualmente Servicios Web) proporciona un medio de acceso remoto basado en canales de comunicación y mensajes de datos. Es importante destacar que esta capa debe ser lo más ligera posible y que no debe incluir nunca 'lógica' de negocio. Hoy por hoy, con las tecnologías actuales hay muchos elementos de una arquitectura que son muy simples de realizar en esta capa y en muchas ocasiones se tiende a incluir en ella propósitos que no le competen.

Capa de Aplicación

Esta capa forma parte de la propuesta de arquitecturas orientadas al Dominio. Define los trabajos que la aplicación como tal debe de realizar y redirige a los objetos del dominio y de infraestructura (persistencia, etc.) que son los que internamente deben resolver los problemas.

Realmente esta capa no debe contener reglas del dominio o conocimiento de la lógica de negocio, simplemente debe realizar tareas de coordinación de aspectos tecnológicos de la aplicación que nunca explicaríamos a un experto del dominio o usuario de negocio. Aquí implementamos la coordinación de la 'fontanería' de la aplicación, como coordinación de transacciones, ejecución de unidades de trabajo, y en definitiva llamadas a tareas necesarias para la aplicación (*software* como tal). Otros aspectos a implementar aquí pueden ser optimizaciones de la aplicación, conversiones de datos/formatos, etc. pero siempre nos referimos solo a la coordinación. El trabajo final se delegará posteriormente a los objetos de las capas inferiores. Esta capa tampoco debe contener estados que reflejen la situación de la lógica de negocio interna pero sí puede tener estados que reflejen el progreso de una tarea de la aplicación con el fin de mostrar dichos progresos al usuario.

Es una capa en algunos sentidos parecida a las capas "*Fachada de Negocio*", pues en definitiva hará de fachada del modelo de Dominio, pero no solamente se encarga de simplificar el acceso al Dominio, hace algo más. Aspectos a incluir en esta capa serían:

- Coordinación de la mayoría de las llamadas a objetos *Repositorios* de la Capa de Persistencia y acceso a datos.

- Agrupaciones/agregaciones de datos de diferentes entidades para ser enviadas de una forma más eficiente (minimizar las llamadas remotas) por la capa superior de servicios web. Estos objetos a envíar son los DTOs, (*Data Transfer Object*) y el código en la capa de aplicación son *DTO-Adapters*.

- Acciones que consolidan o agrupan operaciones del Dominio *dependiendo de las acciones mostradas en la interfaz de usuario,* relacionando dichas acciones con las operaciones de persistencia y acceso a datos.

- Mantenimiento de estados relativos a la aplicación (no estados internos del Dominio).

- Coordinación de acciones entre el Dominio y aspectos de infraestructura. Por ejemplo, la acción de realizar una transferencia bancaria requiere obtener datos de las fuentes de datos haciendo uso de los Repositorios, utilizar posteriormente objetos del dominio con la lógica de negocio de la transferencia (abono y cargo) y a lo mejor finalmente mandar un e-mail a las partes interesadas invocando otro objeto de infraestructura que realice dicho envío del e-mail.

- **Servicios de Aplicación**: Es importante destacar que el concepto de Servicios en una arquitectura *N-Layer* con orientación al dominio, no tiene nada que ver con los Servicios-Web para accesos remotos. Primeramente, el concepto de servicio DDD existe diferentes capas, tanto en las capas de Aplicación, de Dominio e incluso en la de Infraestructura. **El concepto de servicios es simplemente un conjunto de clases donde agrupar comportamientos y métodos de acciones** que no pertenecen a una clase de bajo nivel concreta (como entidades y Servicios del Dominio u otro tipo de clase con identidad propia.). Así pues, los servicios normalmente coordinarán objetos de capas inferiores.

  En cuanto a los 'Servicios de Aplicación', que es el punto actual, estos servicios normalmente coordinan el trabajo de otros servicios de capas inferiores (Servicios de Capas del Dominio o incluso Servicios de capas de Infraestructura transversal). Por ejemplo, un servicio de la capa de aplicación puede llamar a otro servicio de la capa del dominio para que efectúe la lógica de la creación de un pedido en las entidades 'en memoria'. Una vez efectuadas dichas operaciones de negocio por la Capa del Dominio (la mayoría son cambios en objetos en memoria), la capa de aplicación podrá llamar a Repositorios de infraestructura delegando en ellos para que se encarguen de

persistir los cambios en las fuentes de datos. Esto es un ejemplo de coordinación de servicios de capas inferiores.

- **Workflows de Negocio (Opcional):** Algunos procesos de negocio están formados por un cierto número de pasos que deben ejecutarse de acuerdo a unas reglas concretas dependiendo de eventos que se puedan producir en el sistema y, normalmente, con un **tiempo de ejecución total de larga duración** (indeterminado, en cualquier caso), interactuando unos pasos con otros mediante una orquestación dependiente de dichos eventos. Este tipo de procesos de negocio se implementan de forma natural como flujos de trabajo (*workflows*) mediante tecnologías concretas y herramientas de gestión de procesos de negocio especialmente diseñadas para ello.

También esta capa de Aplicación puede ser publicada mediante la capa superior de servicios web, de forma que pueda ser invocada remotamente.

Capa del Dominio

Esta capa es responsable de representar conceptos de negocio, información sobre la situación de los procesos de negocio e implementación de las reglas del dominio. También debe contener los estados que reflejan la situación de los procesos de negocio. **Esta capa, 'Dominio', es el corazón del software.**

Así pues, estos componentes implementan la funcionalidad principal del sistema y encapsulan toda la lógica de negocio relevante (genéricamente llamado lógica del Dominio según nomenclatura DDD). Básicamente suelen ser clases en el lenguaje seleccionado que implementan la lógica del dominio dentro de sus métodos. Siguiendo los patrones de Arquitecturas N-Layer con Orientación al Dominio, esta capa tiene que ignorar completamente los detalles de persistencia de datos. Estas tareas de persistencia deben ser realizadas por las capas de infraestructura y coordinadas por la capa de Aplicación.

Normalmente podemos definir los siguientes elementos dentro de la capa de Dominio:

- **Entidades del Dominio**: Estos objetos son entidades desconectadas (datos + lógica) y se utilizan para alojar y transferir datos de entidades entre las diferentes capas. Pero adicionalmente, una característica fundamental en DDD es que contengan también la lógica del dominio relativo a cada entidad. Por ejemplo, en un abono bancario, la operación de sumar una cantidad de dinero al saldo de una cuenta la debemos realizar con lógica dentro de la propia entidad cuenta. Otros ejemplos son validaciones de datos relacionados con lógica de negocio, campos pre-calculados, relaciones con otras sub-entidades, etc. Estas clases representan al fin y al cabo las entidades de negocio del mundo real, como productos o pedidos. Las entidades de datos que la aplicación utiliza internamente, son en cambio objetos en memoria con datos y cierta lógica relacionada. Si usásemos 'solo los datos' de las entidades sin la lógica de la propia entidad dentro de la misma clase, estaríamos cayendo en el anti-patrón

llamado '*Anemic Domain Model'*, descrito originalmente sobre todo por *Martin Fowler*. Adicionalmente y siguiendo los patrones y principios recomendados, es bueno que estas clases entidad sean también objetos POCO (*Plain Old CLR Objects*), es decir, clases independientes de tecnologías concretas de acceso a datos, con código completamente bajo nuestro control. En definitiva, con este diseño (*Persistence Ignorance*) lo que buscamos es que las clases del dominio 'no sepan nada' de las interioridades de los repositorios ni de las tecnologías de acceso a datos. Cuando se trabaja en las capas del dominio, se debe ignorar cómo están implementados los repositorios.
Las clases entidad se sitúan dentro del dominio, puesto que son entes del dominio e independientes de cualquier tecnología de infraestructura (persistencia de datos, ORMs, etc.). En cualquier caso, las entidades serán objetos flotantes a lo largo de toda o casi toda la arquitectura.
Relativo a DDD, y de acuerdo con la definición de Eric Evans, "*Un objeto primariamente definido por su identidad se le denomina Entidad*". Las entidades son fundamentales en el modelo del Dominio y tienen que ser identificadas y diseñadas cuidadosamente. Lo que en algunas aplicaciones puede ser una entidad, en otras aplicaciones no debe serlo. Por ejemplo, una 'dirección' en algunos sistemas puede no tener una identidad en absoluto, pues puede estar representando solo atributos de una persona o compañía. En otros sistemas, sin embargo, como en una aplicación para una empresa de Electricidad, la dirección de los clientes puede ser muy importante y debe ser una entidad, porque la facturación puede estar ligada directamente con la dirección. En este caso, una dirección tiene que clasificarse como una Entidad del Dominio. En otros casos, como en una aplicación de comercio electrónico, la dirección puede ser simplemente un atributo del perfil de una persona. En este último caso, la dirección no es tan importante y debería clasificarse como un 'Objeto Valor' (En DDD denominado '*Value-Object*').

- **Servicios del Dominio:** En la capas del Dominio, los servicios son básicamente clases agrupadoras de comportamientos y/o métodos con ejecución de lógica del dominio. Estas clases normalmente no deben contener estados relativos al dominio (deben ser clases *stateless*) y serán las clases que coordinen e inicien operaciones compuestas contra las entidades del dominio. Un caso típico de un Servicio del Dominio es que esté relacionado con varias entidades al mismo tiempo. Pero también podemos tener un Servicio que esté encargado de interactuar (obtener, actualizar, etc.) contra una única entidad raíz (la cual sí puede englobar a otros datos relacionados siguiendo el patrón *Aggregate*).

- **Contratos de Repositorios**: Está claro que la implementación de los propios Repositorios no estará en el dominio, puesto que la implementación de los Repositorios no es parte del Dominio sino parte de las capas de Infraestructura (Los Repositorios están ligados a una tecnología de persistencia de datos, como un ORM). Sin embargo, los interfaces o 'contratos' de cómo deben estar construidos dichos Repositorios, si deben formar parte del Dominio. En dichos contratos se especifica qué debe ofrecer cada Repositorio para que funcione y

se integre correctamente con el Dominio, sin importarnos como están implementados por dentro.
Dichos interfaces/contratos si son 'agnósticos' a la tecnología, aun cuando la implementación de los interfaces, por el contrario, si esté ligada a ciertas tecnologías. Así pues, es importante que los interfaces/contratos de los Repositorios estén definidos dentro de las Capas del Dominio. Esto es uno de los puntos recomendados en arquitecturas con orientación al Dominio y está basado en el patrón '*Separated Interface Pattern*' definido por *Martin Fowler*.
Lógicamente, para poder cumplir este punto, es necesario que las '*Entidades del Dominio*' y los '*Value-Objects*' sean POCO; es decir, los objetos encargados de alojar las entidades y datos deben ser también completamente agnósticos a la tecnología de acceso a datos. Hay que tener en cuenta que las entidades del dominio son, al final, los 'tipos' de los parámetros enviados y devueltos por y hacia los Repositorios.

Capa de Infraestructura de Acceso a Datos

Esta capa proporciona la capacidad de persistir datos así como lógicamente acceder a ellos. Pueden ser datos propios del sistema o incluso acceder a datos expuestos por sistemas externos (Servicios Web externos, etc.). Así pues, esta capa de persistencia de datos expone el acceso a datos a las capas superiores, normalmente las capas del dominio. Esta exposición deberá realizarse de una forma desacoplada.

- **Implementación de 'Repositorios':** A nivel genérico, un Repositorio "*Representa todos los objetos de un cierto tipo como un conjunto conceptual*" (Definición de *Eric Evans*). A nivel práctico, un Repositorio será normalmente una clase encargada de realizar las operaciones de persistencia y acceso a datos, estando ligado por lo tanto a una tecnología concreta (p.e. ligado a un ORM como *Entity Framework, NHibernate*, o incluso simplemente *ADO.NET* para un gestor de bases de datos concreto). Haciendo esto centralizamos la funcionalidad de acceso a datos, lo cual hace más directo y sencillo el mantenimiento y configuración de la aplicación.
  Normalmente debemos crear un *Repository* por cada 'Entidad Raíz del Dominio'. Es casi lo mismo que decir que la relación entre un *Repository* y una entidad raíz es una relación 1:1. Las entidades raíz podrán ser a veces aisladas y otras veces la raíz de un '*Aggregate'*, que es un conjunto de entidades '*Object Values*' más la propia entidad raíz.

  El acceso a un Repositorio debe realizarse mediante un interfaz bien conocido, un contrato 'depositado' en el Dominio, de forma que podríamos llegar a sustituir un Repositorio por otro que se implemente con otras tecnologías y, sin embargo, la capa del Dominio no se vería afectada.

  El punto clave de los Repositorios es que deben facilitar al desarrollador el mantenerse centrado en la lógica del modelo del Dominio y esconder por lo tanto la 'fontanería' del acceso a los datos mediante dichos 'contratos' de

repositorios. A este concepto se le conoce también como '*PERSISTENCE IGNORANCE'*, lo cual significa que el modelo del Dominio ignora completamente cómo se persisten o consultan los datos contra las fuentes de datos de cada caso (Bases de datos u otro tipo de almacén).

Por último, es **fundamental diferenciar entre un objeto 'Data Access' (utilizados en muchas arquitecturas tradicionales N-Layer) y un Repositorio**. La principal diferencia radica en que un objeto '*Data Access'* realiza directamente las operaciones de persistencia y acceso a datos contra el almacén (normalmente una base de datos). Sin embargo, un Repositorio 'registra' en memoria (un contexto) las operaciones que se quieren hacer, pero estas no se realizarán hasta que desde la capa de Aplicación se quieran efectuar esas 'n' operaciones de persistencia/acceso en una misma acción, todas a la vez. Esto está basado normalmente en el patrón 'Unidad de Trabajo' o '*Unit of Work*', que se explicará en detalle en el capítulo de 'Capa de Aplicación'. Este patrón o forma de aplicar/efectuar operaciones contra los almacenes, en muchos casos puede aumentar el rendimiento de las aplicaciones, y en cualquier caso, reduce las posibilidades de que se produzcan inconsistencias. También reduce los tiempos de bloqueos en tabla debidos a transacciones.

- **Componentes Base (*Layer SuperType*):** La mayoría de las tareas de acceso a datos requieren cierta lógica común que puede ser extraída e implementada en un componente separado y reutilizable. Esto ayuda a simplificar la complejidad de los componentes de acceso a datos y sobre todo, minimiza el volumen de código a mantener. Estos componentes pueden ser implementados como clases base o clases utilidad (dependiendo del uso) y ser código reutilizado en diferentes proyectos/aplicaciones.

  Este concepto es realmente un patrón muy conocido denominado *'Layered Supertype Pattern'* definido por *Martin Fowler*, que dice básicamente "Si los comportamientos y acciones comunes de un tipo de clases se agrupan en una clase base, esto eliminará muchos duplicados de código y comportamientos". El uso de este patrón es puramente por conveniencia y no distrae de prestar atención al Dominio en absoluto.

  El patrón '*Layered Supertype Pattern'* se puede aplicar a cualquier tipo de capa (Dominios, Infraestructura, etc.), no solamente a los Repositorios.

- **'Modelo de Datos'**: Normalmente los sistemas ORM (como *Entity Framework*) disponen de técnicas de definición del modelo de datos a nivel de diagramas 'entidad-relación', incluso a nivel visual. Esta subcapa deberá contener dichos modelos entidad relación, a ser posible, de forma visual con diagramas.

- **Agentes de Servicios remotos/externos:** Cuando un componente de negocio debe utilizar funcionalidad proporcionada por servicios externos/remotos

(normalmente Servicios Web), se debe implementar código que gestione la semántica de comunicaciones con dicho servicio particular o incluso tareas adicionales como mapeos entre diferentes formatos de datos. Los Agentes de Servicios aíslan dicha idiosincrasia de forma que, manteniendo ciertos interfaces, sería posible sustituir el servicio externo original por un segundo servicio diferente sin que nuestro sistema se vea afectado.

Capas de Infraestructura Transversal/Horizontal

Proporcionan capacidades técnicas genéricas que dan soporte a capas superiores. En definitiva, son 'bloques de construcción' ligados a una tecnología concreta para desempeñar sus funciones.

Existen muchas tareas implementadas en el código de una aplicación que se deben aplicar en diferentes capas. Estas tareas o aspectos horizontales (Transversales) implementan tipos específicos de funcionalidad que pueden ser accedidos/utilizados desde componentes de cualquier capa. Los diferentes tipos/aspectos horizontales más comunes, son: **Seguridad** (Autenticación, Autorización y Validación) y **tareas de gestión de operaciones** (políticas, *logging*, trazas, monitorización, configuración, etc.). Estos aspectos serán detallados en capítulos posteriores.

- **Subcapas de 'Servicios de Infraestructura'**: En las capas de infraestructura transversal también existe el concepto de Servicios. Se encargarán de agrupar acciones de infraestructura, como mandar e-mails, controlar aspectos de seguridad, gestión de operaciones, *logging*, etc. Así pues, estos **Servicios**, agrupan cualquier tipo de actividad de infraestructura transversal ligada a tecnologías específicas.

- **Subcapas de objetos de infraestructura**: Dependiendo del tipo de aspecto de infraestructura transversal, necesitaremos los objetos necesarios para implementarlos, bien sean aspectos de seguridad, trazas, monitorización, envío de e-mails, etc.

Estas capas de 'Infraestructura Transversal' engloban una cantidad muy grande de conceptos diferentes, muchos de ellos relacionados con Calidad de Servicio (*QoS – Quality of Service*) y realmente, cualquier implementación ligada a una tecnología/infraestructura concreta. Es por ello que se definirá en detalle en un capítulo dedicado a estos aspectos transversales.

'Servicios' como concepto genérico disponible en las diferentes Capas

Debido a que los SERVICIOS están presentes en diferentes capas de una Arquitectura DDD, resumimos a continuación en un cuadro especial sobre el concepto de SERVICIO utilizado en DDD.

Tabla 1.- Servicios en Arquitecturas N-Layer Orientadas al Dominio

Servicios en Arquitecturas N-Layer Orientadas al Dominio
Como hemos visto en diferentes Capas (APLICACIÓN, DOMINIO e INFRAESTRUCTURA-TRANSVERSAL), en todas ellas podemos disponer de una sub-capa denominada Servicios. Debido a que es un concepto presente en diferentes puntos, es bueno tener un visón global sobre qué son los 'Servicios' en DDD. Primeramente es importante aclarar, para no confundir conceptos, que los SERVICIOS en DDD no son los SERVICIOS-WEB utilizados para invocaciones remotas. Estos otros SERVICIOS-WEB estarán en una posible capa superior de 'Capa de Servicios Distribuidos' y podrían a su vez publicar las capas inferiores permitiendo acceso remoto a los SERVICIOS-DDD y también a otros objetos de la Capa de Aplicación y de Dominio. Centrándonos en el concepto de SERVICIO en DDD, en algunos casos, los diseños más claros y pragmáticos incluyen operaciones que no pertenecen conceptualmente a objetos específicos de cada capa (p.e. operaciones que no pertenezcan de forma exclusiva a una entidad). En estos casos podemos incluir/agrupar dichas operaciones en SERVICIOS explícitos. Dichas operaciones son intrínsecamente actividades u operaciones, no características de cosas u objetos específicos de cada capa. Pero debido a que nuestro modelo de programación es orientado a objetos, debemos agruparlos también en objetos. A estos objetos les llamamos SERVICIOS. El forzar a dichas operaciones (normalmente operaciones de alto nivel y agrupadoras de otras acciones) a formar parte de objetos naturales de la capa, distorsionaría la definición de los objetos reales de la capa. Por ejemplo, la lógica propia de una entidad debe de estar relacionada con su interior, cosas como validaciones con respecto a sus datos en memoria, o campos calculados, etc., pero no el tratamiento de la propia entidad como un todo. Un 'motor' realiza acciones relativas al uso del motor, no relativas a cómo se fabrica dicho motor. Así mismo, la lógica perteneciente a una clase entidad no debe encargarse de su propia persistencia y almacenamiento. Un SERVICIO es una operación o conjunto de operaciones ofrecidas como un interfaz que simplemente está disponible en el modelo, sin encapsular estados. La palabra "Servicio" del patrón SERVICIO precisamente hace hincapié en lo que ofrece: "*Qué puede hacer y qué acciones ofrece al cliente que lo consuma y enfatiza la relación con otros objetos de cada capa*".

A algunos SERVICIOS (sobre todo los de más alto nivel, en la Capa de Aplicación y/o algunos servicios del dominio coordinadores de lógica de negocio) se les suele nombrar con nombres de Actividades, no con nombres de objetos. Están por lo tanto relacionados con verbos de los Casos de Uso del análisis, no con sustantivos (objetos), aun cuando puede tener una definición abstracta de una operación concreta (Por ejemplo, un Servicio-Transferencia relacionado con la acción/verbo 'Transferir Dinero de una cuenta bancaria a otra').

Los servicios no deben tener estados (deben ser *stateless*). Esto no implica que la clase que lo implementa tenga que ser estática, podrá ser perfectamente una clase instanciable. Que un SERVICIO sea *stateless* significa que un programa cliente puede hacer uso de cualquier instancia de un servicio sin importar su historia individual como objeto.

Adicionalmente, la ejecución de un SERVICIO hará uso de información que es accesible globalmente y puede incluso cambiar dicha información (es decir, normalmente provoca cambios globales). Pero el servicio no contiene estados que pueda afectar a su propio comportamiento, como si tienen por ejemplo las entidades.

A modo de aclaración mostramos como particionar diferentes Servicios en diferentes capas en un escenario bancario simplificado:

APLICACIÓN	*Servicio de **Aplicación** de 'BankingService' (Operaciones Bancarias)* - *Asimila y convierte formatos de datos de entrada (Como conversiones de datos XML)* - *Proporciona datos de la transferencia a la Capa de Dominio para que sea allí realmente procesada la lógica de negocio.* - *Coordina/invoca a los objetos de persistencia (Repositorios) de la capa de infraestructura, para persistir los cambios realizados en las entidades e cuentas bancarias, por la capa del dominio.* - *Decide si se envía notificación (e-mail al usuario) utilizando servicios de infraestructura transversal.* - *En definitiva, implementa toda la 'coordinación de la fontanería tecnológica'*

	(como uso de transacciones y Unit of Work) para que la Capa de Dominio quede lo más limpia posible y exprese mejor y muy claramente su lógica.
DOMINIO	*Servicio de **Dominio** de ‘Transferencia-Bancaria’ (Verbo Transferir Fondos)* - *Coordina el uso de los objetos entidad como ‘CuentaBancaria’ y otros objetos del Dominio bancario.* - *Proporciona confirmación del resultado de las operaciones de negocio.*
INFRAESTRUCTURA-TRANSVERSAL	*Servicio de **Infraestructura Transversal** de ‘Envío Notificaciones’ (Verbo Enviar/Notificar)* - *Envía un correo electrónico, mensaje SMS u otro tipo de comunicación requerido por la aplicación*

De todo lo explicado hasta este punto en el presente capítulo, se desprende la primera regla a cumplir en un desarrollo de aplicación empresarial siguiendo esta guía de Arquitectura Marco:

Tabla 2.- Regla de Diseño D1

Regla Nº: D1.	**El diseño de <u>arquitectura lógica interna de una aplicación</u> se realizará siguiendo el modelo de arquitectura de aplicaciones en N-Capas (*N-Layered*) con Orientación al Dominio y tendencias y patrones DDD (*Domain Driven Design*)**

○ **<u>Normas</u>**

- Por regla general, esta regla deberá aplicarse en casi el 100% de aplicaciones empresariales complejas, con un cierto volumen y propietarias de mucha lógica de Dominio.

✓ **Cuándo SÍ implementar una arquitectura N-Capas con Orientación al Dominio**

- Deberá implementarse en las aplicaciones empresariales complejas cuya lógica de negocio cambie bastante y la aplicación vaya a sufrir cambios y mantenimientos posteriores durante una vida de aplicación, como mínimo, relativamente larga.

✘ **Cuándo NO implementar una arquitectura N-Capas DDD**

- En aplicaciones pequeñas que una vez finalizadas se prevén pocos cambios, la vida de la aplicación será relativamente corta y donde prima la velocidad en el desarrollo de la aplicación. En estos casos se recomienda implementar la aplicación con tecnologías RAD (como puede ser '*Microsoft RIA Services*'), aunque tendrá la desventaja de implementar componentes más fuertemente acoplados, la calidad resultante de la aplicación será peor y el coste futuro de mantenimiento probablemente será mayor dependiendo de si la aplicación continuará su vida con un volumen grande de cambios o no.

Ventajas del uso de Arquitecturas N-Capas

→ Desarrollo estructurado, homogéneo y similar de las diferentes aplicaciones de una organización.

→ Facilidad de mantenimiento de las aplicaciones pues los diferentes tipos de tareas están siempre situados en las mismas áreas de la arquitectura.

→ Fácil cambio de tipología en el despliegue físico de una aplicación (2-Tier, 3-Tier, etc.), pues las diferentes capas pueden separarse físicamente de forma fácil.

Desventajas del uso de Arquitecturas N-Capas

→ En el caso de aplicaciones muy pequeñas, estamos añadiendo una complejidad excesiva (capas, desacoplamiento, etc.). Pero este caso es muy poco probable en aplicaciones empresariales con cierto nivel.

Referencias

Eric Evans: Libro "Domain-Driven Design: Tackling Complexity in the Heart of Software"

Martin Fowler: Definición del 'Domain Model Pattern' y Libro "Patterns of Enterprise Application Architecture"

Jimmy Nilson: Libro "Applying Domain-Driven-Desing and Patterns with examples in C# and .NET"

SoC - Separation of Concerns principle:
http://en.wikipedia.org/wiki/Separation_of_concerns

EDA - Event-Driven Architecture: SOA Through the Looking Glass – "The Architecture Journal"

EDA - Using Events in Highly Distributed Architectures – "The Architecture Journal"

Sin embargo, aunque estas son las capas inicialmente propuestas para cubrir un gran porcentaje de aplicaciones **N-Layered**, la arquitectura base está abierta a la implementación de nuevas capas y personalizaciones necesarias para una aplicación dada (por ejemplo capa EAI para integración con aplicaciones externas, etc.).

Así mismo, tampoco es obligatoria la implementación completa de las capas de componentes propuestas. Por ejemplo, en algunos casos podría no implementarse la capa de Servicios-Web por no necesitar implementar accesos remotos, etc.

2.3.- Desacoplamiento entre componentes

Es fundamental destacar que no solo se deben de delimitar los componentes de una aplicación entre diferentes capas. Adicionalmente, también debemos tener especial atención en cómo interaccionan unos componentes/objetos con otros, es decir, cómo se consumen y en especial cómo se instancian unos objetos desde otros.

En general, este desacoplamiento debería realizarse entre todos los objetos (con lógica de ejecución y dependencias) pertenecientes a las diferentes capas, pues existen

ciertas capas las cuales nos puede interesar mucho el que se integren en la aplicación de una forma desacoplada. Este es el caso de la mayoría de capas de Infraestructura (ligadas a unas tecnologías concretas), como puede ser la propia capa de persistencia de datos, que podemos haber ligado a una tecnología concreta de ORM o incluso a un acceso a *backend* externo concreto (p.e. ligado a accesos a un Host, ERP o cualquier otro *backend* empresarial). En definitiva, para poder integrar esa capa de forma desacoplada, no debemos instanciar directamente sus objetos (p.e., no instanciar directamente los objetos Repositorio o cualquier otro relacionado con una tecnología concreta, de la infraestructura de nuestra aplicación).

Pero la esencia final de este punto, realmente trata del desacoplamiento entre cualquier tipo/conjunto de objetos. Bien sean conjuntos de objetos diferentes dentro del propio Dominio (p.e. para un país, cliente o tipología concreta, poder inyectar unas clases específicas de lógica de negocio), o bien, en los componentes de Capa de presentación poder simular la funcionalidad de Servicios-Web, o en la Capa de Persistencia poder también simular otros Servicios-Web externos y en todos esos casos realizarlo de forma desacoplada para poder cambiar de la ejecución real a la simulada o a otra ejecución real diferente, con el menor impacto. En todos esos ejemplos tiene mucho sentido un desacoplamiento de por medio.

En definitiva, es conseguir un '*state of the art'* del diseño interno de nuestra aplicación: *"Tener preparada toda la estructura de la Arquitectura de tu aplicación de forma desacoplada y en cualquier momento poder inyectar funcionalidad para cualquier área o grupo de objetos, no tiene por qué ser solo entre capas diferentes".*

Un enfoque exclusivo de "desacoplamiento entre capas" probablemente no es el más correcto. El ejemplo de conjuntos de objetos diferentes a inyectar dentro del propio Dominio, que es una única capa (p.e. para un país, cliente o tipología concreta, un módulo incluso vertical/funcional), clarifica bastante.

En la aplicación ejemplo anexa a esta Guía de Arquitectura hemos optado por realizar desacoplamiento entre todos los objetos de las capas internas de la aplicación, porque ofrece muchas ventajas y así mostramos la mecánica completa.

Las técnicas de desacoplamiento están basadas en el **Principio de Inversión de Dependencias**, el cual establece una forma especial de desacoplamiento donde se invierte la típica relación de dependencia que se suele hacer en orientación a objetos la cual decía que las capas de alto nivel deben depender de las Capas de más bajo nivel. El propósito es conseguir disponer de capas de alto nivel que sean independientes de la implementación y detalles concretos de las capas de más bajo nivel, y por lo tanto también, independientes de las tecnologías subyacentes.

El **Principio de Inversión de Dependencias** establece:

A. Las capas de alto nivel no deben depender de las capas de bajo nivel. Ambas capas deben depender de abstracciones (Interfaces)

B. Las abstracciones no deben depender de los detalles. Son los Detalles (Implementación) los que deben depender de las abstracciones (Interfaces).

El objetivo del principio de inversión de dependencias es desacoplar los componentes de alto nivel de los componentes de bajo nivel de forma que sea posible llegar a reutilizar los mismos componentes de alto nivel con diferentes implementaciones de componentes de bajo nivel. Por ejemplo, poder reutilizar la misma Capa de Dominio con diferentes Capas de Infraestructura que implementen diferentes tecnologías ("diferentes detalles") pero cumpliendo los mismos interfaces (abstracciones) de cara a la Capa de Dominio.

Los contratos/interfaces definen el comportamiento requerido a los componentes de bajo nivel por los componentes de alto nivel y además dichos contratos/interfaces deben existir en los *assemblies* de alto nivel.

Cuando los componentes de bajo nivel implementan los interfaces/contratos a cumplir (que se encuentran en las capas de alto nivel), significa que los componentes/capas de bajo nivel son las que dependen, a la hora de compilar, de los componentes de alto nivel, invirtiendo la tradicional relación de dependencia. Por eso se llama "*Inversión de Dependencias*".

Existen varias técnicas y patrones que se utilizan para facilitar el 'aprovisionamiento' de la implementación elegida de las capas/componentes de bajo nivel, como son *Plugin, Service Locator, Dependency Injection e IoC (Inversion of Control).*

Básicamente, las técnicas principales que proponemos utilizar para habilitar el desacoplamiento entre capas, son:

- Inversión de control (IoC)
- Inyección de dependencias (DI)
- Interfaces de Servicios Distribuidos (para consumo/acceso remoto a capas)

El uso correcto de estas técnicas, gracias al desacoplamiento que aportan, potencia los siguientes puntos:

- Posibilidad de sustitución, en tiempo de ejecución, de capas/módulos actuales por otros diferentes (con mismos interfaces y similar comportamiento), sin que impacte a la aplicación. Por ejemplo, puede llegar a sustituirse en tiempo de ejecución un módulo que accede a una base de datos por otro que accede a un sistema externo tipo HOST o cualquier otro tipo de sistema, siempre y cuando cumplan unos mismos interfaces. No sería necesario el añadir el nuevo módulo, especificar referencias directas y recompilar nuevamente la capa que lo consume.

- Posibilidad de uso de **STUBS/MOLES y MOCKS** en pruebas: Es realmente un escenario concreto de cambio de un módulo por otro. En este caso consiste por ejemplo, en sustituir un módulo de acceso a datos reales (a bases de datos o cualquier otra fuente de datos) por un módulo con interfaces similares pero que simplemente simula que accede a las fuentes de datos. Mediante la inyección de dependencias puede realizarse este cambio incluso en tiempo de ejecución, sin llegar a tener que recompilar la solución.

2.4.- Inyección de dependencias e Inversión de control

Patrón de Inversión de Control (IoC): Delegamos a un componente o fuente externa, la función de seleccionar un tipo de implementación concreta de las dependencias de nuestras clases. En definitiva, este patrón describe técnicas para soportar una arquitectura tipo 'plug-in' donde los objetos pueden buscar instancias de otros objetos que requieren y de los cuales dependen.

Patrón Inyección de Dependencias (*Dependency Injection, DI*): Es realmente un caso especial de IoC. Es un patrón en el que se suplen objetos/dependencias a una clase en lugar de ser la propia clase quien cree los objetos/dependencias que necesita. El término fue acuñado por primera vez por *Martin Fowler*.

Entre las diferentes capas no debemos de instanciar de forma explícita las dependencias. Para conseguir esto, se puede hacer uso de una clase base o un interfaz (nos parece más claro el uso de interfaces) que defina una abstracción común que pueda ser utilizada para inyectar instancias de objetos en componentes que interactúen con dicho interfaz abstracto compartido.

Para dicha inyección de objetos, inicialmente se podría hacer uso de un "Constructor de Objetos" (*Patrón Factory*) que crea instancias de nuestras dependencias y nos las proporciona a nuestro objeto origen, durante la creación del objeto y/o inicialización. Pero, la forma más potente de implementar este patrón es mediante un "*Contenedor DI*" (En lugar de un "Constructor de Objetos" creado por nosotros). El contenedor DI inyecta a cada objeto las dependencias/objetos necesarios según las relaciones o registro plasmado bien por código o en ficheros XML de configuración del "Contenedor DI".

Típicamente este contenedor DI es proporcionado por un *framework* externo a la aplicación (como *Unity, Castle-Windsor, Spring.NET,* etc.), por lo cual en la aplicación también se utilizará Inversión de Control al ser el contenedor (almacenado en una biblioteca) quien invoque el código de la aplicación.

Los desarrolladores codificarán contra un interfaz relacionado con la clase y usarán un contenedor que inyectará las instancias de los objetos dependientes en la clase en base al interfaz o clase declarada de los objetos dependientes. Las técnicas de inyección de instancias de objetos son 'inyección de interfaz', 'inyección de constructor', 'inyección de propiedad' (*setter*), e 'inyección de llamada a método'.

Cuando la técnica de 'Inyección de Dependencias' se utiliza para desacoplar objetos de nuestras capas, el diseño resultante aplicará por lo tanto el "*Principio de Inversión de Dependencias".*

Un escenario interesante de desacoplamiento con IoC es internamente dentro de la Capa de Presentación, para poder realizar un *mock o stub/mole* de una forma aislada y configurable de los componentes en arquitecturas de presentación tipo MVC y MVVM, donde para una ejecución rápida de pruebas unitarias podemos querer simular un consumo de Servicio Web cuando realmente no lo estamos consumiendo, sino simulando.

Y por supuesto, **la opción más potente relativa al desacoplamiento es hacer uso de IoC y DI entre prácticamente todos los objetos pertenecientes a las capas de la arquitectura, esto nos permitirá en cualquier momento inyectar simulaciones de comportamiento o diferentes ejecuciones reales cambiándolo en tiempo de ejecución y/o configuración.**

En definitiva, **los contenedores IoC y la Inyección de dependencias añaden flexibilidad y conllevan a 'tocar' el menor código posible según avanza el proyecto.** Añaden comprensión y mantenibilidad del proyecto.

Tabla 3.- Inyección de Dependencias (DI) y Desacoplamiento entre objetos como 'Mejor Práctica'

Inyección de Dependencias (DI) y Desacoplamiento entre objetos como 'Mejor Práctica'
El **principio de 'Única responsabilidad' (*Single Responsability Principle*)** establece que cada objeto debe de tener una única responsabilidad. El concepto fue introducido por *Robert C. Martin*. Se establece que una responsabilidad es una razón para cambiar y concluye diciendo que una clase debe tener una y solo una razón para cambiar. Este principio está ampliamente aceptado por la industria del desarrollo y en definitiva promueve el diseño y desarrollo de clases pequeñas con una única responsabilidad. Esto está directamente relacionado con el número de dependencias (objetos de los que depende) cada clase. Si una clase tiene una única responsabilidad, sus métodos normalmente deberán tener pocas dependencias con otros objetos. Si hay una clase con muchísimas dependencias (por ejemplo 15 dependencias), esto nos estaría indicando lo que típicamente se dice como un 'mal olor' del código. Precisamente, haciendo uso de Inyección de dependencias en el constructor, por sistema nos vemos obligados a declarar todas las dependencias de objetos en el constructor y en dicho ejemplo veríamos muy claramente que esa clase en concreto parece que no sigue el principio de '*Single Responsability*', pues es bastante raro que la clase tenga una única responsabilidad y sin embargo en su constructor veamos declaradas 15 dependencias. Así pues, DI es también una forma de guía que nos conduce a realizar buenos diseños e implementaciones en desarrollo, además de ofrecernos un desacoplamiento que podemos utilizar para inyectar diferentes ejecuciones de forma transparente.

Mencionar también que es factible diseñar e implementar una Arquitectura Orientada al Dominio (siguiendo patrones con tendencias DDD) sin implementar técnicas de desacoplamiento (Sin IoC ni DI). No es algo 'obligatorio', pero sí que

favorece mucho el aislamiento del Dominio con respecto al resto de capas, lo cual si es un objetivo primordial en DDD. La inversa también es cierta, es por supuesto también factible utilizar técnicas de desacoplamiento (*IoC y Dependency Injection*) en Arquitecturas no Orientadas al Dominio. En definitiva, hacer uso de IoC y DI, es una filosofía de diseño y desarrollo que nos ayuda a crear un código mejor diseñado y que favorece, como decíamos el principio de '*Single Responsability'*.

Los contenedores IoC y la inyección de dependencias favorecen y facilitan mucho el realizar correctamente Pruebas Unitarias y Mocking. Diseñar una aplicación de forma que pueda ser probada de forma efectiva con Pruebas Unitarias nos fuerza a realizar 'un buen trabajo de diseño' que deberíamos estar haciendo si realmente sabemos qué estamos haciendo en nuestra profesión.

Los interfaces y la inyección de dependencias ayudan a hacer que una aplicación sea extensible (tipo *pluggable*) y eso a su vez ayuda también al *testing*. Podríamos decir que esta facilidad hacia el testing es un efecto colateral 'deseado', pero no el más importante proporcionado por IoC y DI.

Sin embargo, IoC y DI no son solo para favorecer las Pruebas Unitarias, como remarcamos aquí:

Tabla 4.- IoC y DI no son solo para favorecer las Pruebas Unitarias

¡¡IoC y DI no son solo para favorecer las Pruebas Unitarias!!
Esto es fundamental. ¡La Inyección de Dependencias y los contenedores de Inversión de Control no son solo para favorecer el *Testing* de Pruebas Unitarias e Integración! Decir eso sería como decir que el propósito principal de los interfaces es facilitar el *testing*. Nada más lejos de la realidad. DI e IoC tratan sobre desacoplamiento, mayor flexibilidad y disponer de un punto central donde ir que nos facilite la mantenibilidad de nuestras aplicaciones. El *Testing* es importante, pero no es la primera razón ni la más importante por la que hacer uso de Inyección de Dependencias ni IoC.

Otro aspecto a diferenciar es dejar muy claro que DI y los contenedores IoC no son lo mismo.

Tabla 5.- Diferenciamiento entre DI e IoC

DI e IoC son cosas diferentes
Hay que tener presente que DI e IoC son cosas diferentes. DI (Inyección de dependencias mediante constructores o propiedades) puede sin duda ayudar al *testing* pero el aspecto útil principal de ello es que guía a la aplicación hacia el **Principio de Única Responsabilidad** y también normalmente hacia el principio de **'Separación de Preocupaciones/Responsabilidades'** (*Separation Of*

Concerns Principle). Por eso, DI es una técnica muy recomendada, una mejor práctica en el diseño y desarrollo de software.

Debido a que implementar DI por nuestros propios medios (por ejemplo con clases *Factory*) puede llegar a ser bastante farragoso, se usan contenedores IoC para proporcionar flexibilidad a la gestión del grafo de dependencias de objetos.

Tabla 6.- Regla de Diseño Nº D2

Regla Nº: D2.	**El consumo y comunicación entre los diferentes objetos pertenecientes a las capas de la arquitectura deberá ser desacoplado, implementando los patrones de 'Inyección de dependencias' (DI) e 'Inversión de Control' (IoC).**

Normas

- Por regla general, esta regla deberá aplicarse en todas la arquitecturas N-Capas de aplicaciones medianas/grandes. *Por supuesto, debe de realizarse entre los objetos cuya función mayoritaria es la lógica de ejecución (de cualquier tipo) y que tienen dependencias con otros objetos. Un ejemplo claro son los Servicios, Repositorios, etc. No tiene mucho sentido hacerlo con las propias clases de Entidades.*

Cuándo SÍ implementar 'Inyección de dependencias' e 'Inversión de Control'

- Deberá implementarse en prácticamente la totalidad de las aplicaciones empresariales N-Capas que tengan un volumen mediano/grande. Es especialmente útil entre las capas del Dominio y las de Infraestructura así como en la capa de presentación junto con patrones tipo MVC y M-V-VM.

Cuándo NO implementar 'Inyección de dependencias' e 'Inversión de Control'

- A nivel de proyecto, normalmente, no se podrá hacer uso de DI e IoC en aplicaciones desarrolladas con tecnologías RAD (*Rapid Application Development*) que no llegan a implementar realmente una aplicación N-Capas flexible y no hay posibilidad de introducir este tipo de desacoplamiento. Esto pasa habitualmente en aplicaciones pequeñas.

- **A nivel de objetos, en las clases que son 'finales' o no tienen dependencias (como las ENTIDADES), no tiene sentido hacer uso de**

IoC.

<u>Ventajas del uso de 'Inyección de dependencias' e 'Inversión de Control'</u>

→ Posibilidad de sustitución de Capas/bloques, en tiempo de ejecución.

→ Facilidad de uso de STUBS/MOCKS/MOLES para el *Testing* de la aplicación.

→ Añaden flexibilidad y conllevan a 'tocar' el menor código posible según avanza el proyecto.

→ Añaden comprensión y mantenibilidad al proyecto.

<u>Desventajas del uso de 'Inyección de dependencias' e 'Inversión de Control'</u>

→ Si no se conocen las técnicas IoC y DI, se añade cierta complejidad inicial en el desarrollo de la aplicación, pero una vez comprendidos los conceptos, realmente merece la pena en la mayoría de las aplicaciones, ya que añade mucha flexibilidad y finalmente calidad de software.

Referencias

Inyección de Dependencias: *MSDN - http://msdn.microsoft.com/enus/library/cc707845.aspx*

Inversión de Control: *MSDN - http://msdn.microsoft.com/en-us/library/cc707904.aspx*

Inversion of Control Containers and the Dependency Injection pattern *(By Martin Fowler) - http://martinfowler.com/articles/injection.html*

2.5.- Módulos

En las aplicaciones grandes y complejas, el modelo de Dominio tiende a crecer extraordinariamente. El modelo llega a un punto donde es complicado hablar sobre ello como 'un todo', y puede costar bastante entender bien todas sus relaciones e interacciones entre todas sus áreas. Por esa razón, se hace necesario organizar y *particionar* el modelo en diferentes módulos. Los módulos se utilizan como un método de organización de conceptos y tareas relacionadas (normalmente bloques de negocio diferenciados) para reducir la complejidad desde un punto de vista externo.

El concepto de módulo es realmente algo utilizado en el desarrollo de software desde sus orígenes. Es más fácil ver la foto global de un sistema completo si lo subdividimos en diferentes módulos verticales y después en las relaciones entre dichos módulos. Una vez que se entienden las interacciones entre dichos módulos, es más sencillo focalizarse en más detalle de cada uno de ellos. Es una forma simple y eficiente de gestionar la complejidad. El lema "Divide y vencerás" es la frase que mejor lo define.

Un buen ejemplo de división en módulos son la mayoría de los ERPs. Normalmente están divididos en módulos verticales, cada uno de ellos responsable de un área de negocio específico. Ejemplos de módulos de un ERP podrían ser: Nómina, Gestión de Recursos Humanos, Facturación, Almacén, etc.

Otra razón por la que hacer uso de módulos está relacionada con la calidad del código. Es un principio aceptado por la industria el hecho de que **el código debe tener un alto nivel de cohesión y un bajo nivel de acoplamiento.** Mientras que la cohesión empieza en el nivel de las clases y los métodos, también puede aplicarse a nivel de módulo. Es recomendable, por lo tanto, agrupar las clases relacionadas en módulos, de forma que proporcionemos la máxima cohesión posible. Hay varios tipos de cohesión. Dos de las más utilizadas son "Cohesión de Comunicaciones" y "Cohesión Funcional". La cohesión relacionada con las comunicaciones tiene que ver con partes de un módulo que operan sobre los mismos conjuntos de datos. Tiene todo el sentido agruparlo, porque hay una fuerte relación entre esas partes de código. Por otro lado, la cohesión funcional se consigue cuando todas las partes de un módulo realizan una tarea o conjunto de tareas funcionales bien definidas. Esta cohesión es el mejor tipo.

Así pues, el uso de módulos en un diseño es una buena forma de aumentar la cohesión y disminuir el acoplamiento. Habitualmente los módulos se dividirán y repartirán las diferentes áreas funcionales diferenciadas y que no tienen una relación/dependencia muy fuerte entre ellas. Sin embargo, normalmente tendrá que existir algún tipo de comunicación entre los diferentes módulos, de forma que deberemos definir también interfaces para poder comunicar unos módulos con otros. En lugar de llamar a cinco objetos de un módulo, probablemente es mejor llamar a un interfaz (p.e. un Servicio DDD) del otro módulo que agrega/agrupa un conjunto de funcionalidad. Esto reduce también el acoplamiento.

Un bajo acoplamiento entre módulos reduce la complejidad y mejora sustancialmente la mantenibilidad de la aplicación. Es mucho más sencillo también entender cómo funciona un sistema completo, cuando tenemos pocas conexiones entre módulos que realizan tareas bien definidas. En cambio, si tenemos muchas conexiones de unos módulos a otros es mucho más complicado entenderlo, y si es necesario tenerlo así, probablemente debería ser un único módulo. Los módulos deben ser bastante independientes unos de otros.

El nombre de cada módulo debería formar parte del '*Lenguaje Ubicuo*' de DDD, así como cualquier nombre de entidades, clases, etc. Para más detalles sobre qué es el '*Lenguaje Ubicuo*' en DDD, leer documentación sobre DDD como el libro de *Domain-Driven Design* de Eric Evans.

A continuación mostramos el esquema de arquitectura propuesta pero teniendo en cuenta diferentes posibles módulos de una aplicación:

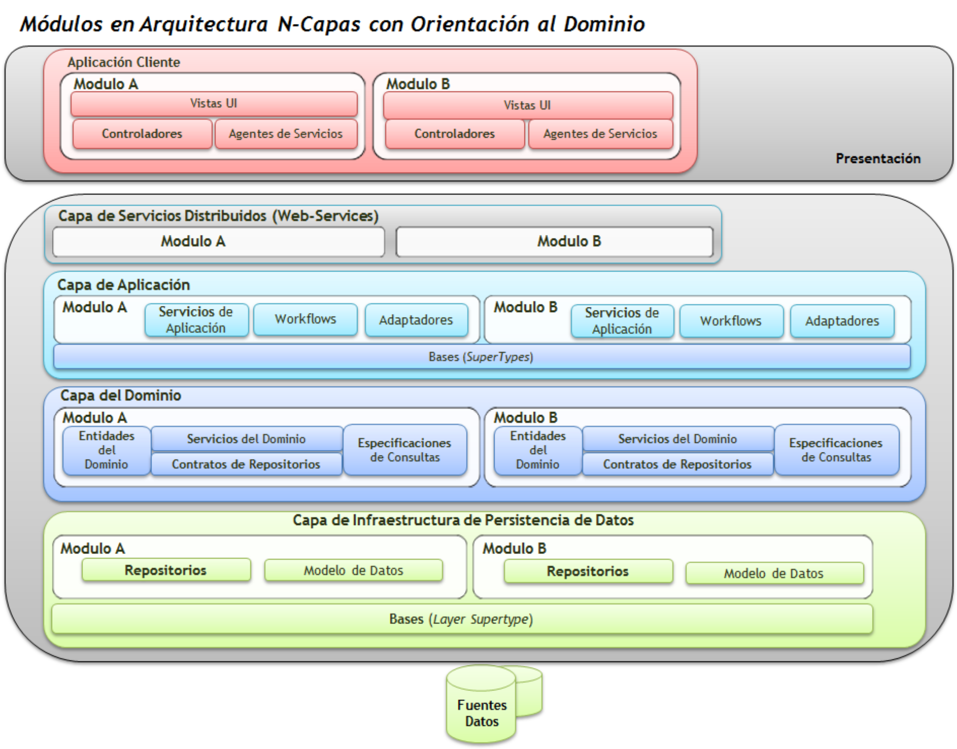


Figura 9.- Módulos en Arquitectura N-Capas con Orientación al Dominio

A nivel de interfaz de usuario, el problema que surge cuando hay diferentes grupos de desarrollo trabajando en los diferentes módulos es que, al final, la capa de presentación (la aplicación cliente) es normalmente solo una y los cambios a realizar en ella por unos grupos de desarrollo pueden molestar/estorbar a los cambios a hacer por otros grupos de desarrollo.

Debido a esto, los módulos tienen mucho que ver con el concepto de aplicaciones compuestas ***('Composite Applications'),*** donde diferentes grupos de desarrollo pueden estar trabajando sobre la misma aplicación pero de una forma independiente, cada equipo de desarrollo en un módulo diferente. Pero finalmente todo se tiene que integrar en el mismo interfaz de usuario. Para que esa integración visual sea mucho menos traumática, es deseable hacer uso de conceptos de *'Composite Applications'*, es decir, definir interfaces concretos que cada módulo visual debe cumplir (áreas de menús, áreas de contenido, carga/descarga de módulos visuales a partir de un punto configurable de la aplicación, etc.), de forma que sea una integración muy automatizada y reglada y no algo traumático al hacer la integración de los diferentes módulos en una única aplicación cliente.

Tabla 7.- Regla de Diseño Nº D3

Regla Nº: D3.	**Definir y diseñar Módulos de Aplicación que engloben áreas funcionales diferenciadas.**

○ **Normas**

- Por regla general, esta regla deberá aplicarse en la mayoría de las aplicaciones con cierto volumen y áreas funcionales diferenciadas.

✓ **Cuándo SÍ diseñar e implementar Módulos**

- Deberá implementarse en prácticamente la totalidad de las aplicaciones empresariales que tengan un volumen mediano/grande y sobre todo donde se pueda diferenciar diferentes áreas funcionales que sean bastante independientes entre ellas.

✗ **Cuándo NO diseñar e implementar Módulos**

- En aplicaciones donde se disponga de una única área funcional muy cohesionada entre ella y sea muy complicado separarlo en módulos funcionales independientes y desacoplados a nivel funcional.

Ventajas del uso de 'Módulos'

→ El uso de módulos en un diseño es una buena forma de aumentar la cohesión

y disminuir el acoplamiento.

→ Un bajo acoplamiento entre módulos reduce la complejidad y mejora sustancialmente la mantenibilidad de la aplicación.

<u>Desventajas del uso de 'Módulos'</u>

→ Si las entidades de un hipotético módulo tienen muchas relaciones con otras entidades de otro/s módulos, probablemente debería ser un único módulo.

→ Cierta inversión en tiempo inicial añadido de diseño que obliga a definir interfaces de comunicación entre unos módulos y otros. Sin embargo, siempre que encaje bien la definición y separación de módulos (existen áreas funcionales diferenciadas), será muy beneficioso para el proyecto.

Referencias

***Modules:** Libro DDD – Eric Evans*

Microsoft - Composite Client Application Library: http://msdn.microsoft.com/en-us/library/cc707819.aspx

2.6.- Subdivisión de modelos y contextos de trabajo

En esta sección veremos cómo trabajar con modelos de gran tamaño, expondremos técnicas para mantener la coherencia de los modelos mediante la división de un modelo de gran tamaño en varios modelos más pequeños con fronteras bien definidas. En esta sección nos centraremos en los bounded context. Es vital tener claro que un bounded context no es lo mismo que un contexto de un ORM tipo *Entity Framework* o sesiones de *NHibernate*, sino que representa un concepto completamente distinto, de contexto de trabajo de un grupo de desarrollo, como veremos a continuación.

2.7.- Bounded Contexts

En aplicaciones de gran tamaño y complejidad nuestros modelos crecen muy rápidamente en términos de número de elementos y relaciones entre los mismos. Mantener la coherencia en modelos tan grandes es muy complicado debido tanto al tamaño de los mismos como a la cantidad de personas trabajando al mismo tiempo en ellos. Es muy fácil que dos personas tengan interpretaciones distintas de un mismo concepto, o que repliquen un concepto en otro objeto por no saber que dicho concepto está ya implementado en otro objeto. Para solucionar estos problemas debemos poner un límite al tamaño de los modelos definiendo un contexto dentro del cual dichos modelos son válidos.

La idea de tener un modelo único para todo el sistema es tentadora pero irrealizable debido a que mantener la coherencia dentro de un modelo tan grande es casi imposible y no merece la pena en términos de coste. De hecho, la primera pregunta que debemos hacernos al afrontar el desarrollo de un modelo de gran tamaño es ¿Necesitamos total integración entre cada una de las funcionalidades de nuestro sistema? La respuesta a esta pregunta será no en el 90% de los casos.

Por tanto, los modelos grandes los vamos a separar en varios modelos de menor tamaño, estableciendo que dado un elemento determinado de nuestro sistema, este solo tiene sentido dentro del contexto (o submodelo) donde está definido. Nos centraremos en mantener la coherencia dentro de estos contextos y trataremos aparte las relaciones entre contextos. Los contextos son particiones del modelo destinadas a mantener la coherencia, no una simple partición funcional del mismo. Las estrategias para definir contextos pueden ser múltiples, como por ejemplo dividir en contextos por equipos de trabajo (la idea es fomentar la comunicación y la integración continua dentro de un contexto), por funcionalidades de alto nivel del sistema (uno o varios módulos funcionales), etc. Por ejemplo, en un proyecto donde estamos construyendo un sistema nuevo que debe funcionar en paralelo con un sistema en mantenimiento, está claro que el sistema antiguo tiene su contexto, y que no queremos que nuestro nuevo sistema esté en su mismo contexto, ya que esto influiría en el diseño de nuestro nuevo sistema. Otro posible ejemplo es la existencia de un algoritmo optimizado para algún tipo cálculo donde se utiliza un modelo completamente distinto, como por ejemplo cualquier tipo de cálculo matemático complejo que queramos realizar sobre los elementos de nuestro modelo.

Establecer contextos dentro de un sistema tiene el inconveniente de que perdemos la visión global del mismo, esto provoca que cuando dos contextos tienen que comunicarse para implementar una funcionalidad tiendan a mezclarse. Por este motivo es fundamental definir simultáneamente a los contextos un mapa de contextos, donde se establecen claramente los distintos contextos existentes en el sistema y las relaciones entre los mismos. De esta forma obtenemos las ventajas de coherencia y cohesión que

nos ofrecen los contextos y preservamos la visión global del sistema estableciendo claramente las relaciones entre contextos.

2.8.- Relaciones entre contextos

Las distintas relaciones que se dan entre dos o más contextos dependen fundamentalmente del grado de comunicación que exista entre los distintos equipos de cada contexto y del grado de control que se tenga de los mismos. Por ejemplo, puede ocurrir que no podamos realizar modificaciones en un contexto, como puede ser el caso de un sistema en producción o descontinuado, o puede ocurrir que nuestro sistema se apoye en otros sistemas para funcionar. A continuación veremos algunas relaciones que típicamente se dan entre contextos, pero es importante entender que no debemos forzar estas relaciones entre contextos en nuestro sistema a no ser que se presenten de forma natural.

2.8.1.- Shared Kernel

Cuando tenemos dos o más contextos en los que trabajan equipos que pueden comunicarse de forma fluida, es interesante establecer una responsabilidad compartida sobre los objetos que ambos contextos utilizan para relacionarse con el otro contexto. Estos objetos pasan a formar lo que se denomina el shared kernel o núcleo compartido de ambos contextos, y queda establecido que para realizar una modificación en cualquier objeto del shared kernel se requiere la aprobación de los equipos de todos los contextos implicados. Es recomendable crear conjuntamente entre todos los equipos de los contextos implicados pruebas unitarias para cada objeto del shared kernel, de forma que el comportamiento del shared kernel quede completamente definido. Favorecer la comunicación entre los distintos equipos es crítico, por lo que una buena práctica es hacer circular a algunos miembros de cada equipo por los equipos de los otros contextos, de manera que el conocimiento acumulado en un contexto se transmita al resto.

2.8.2.- Customer/Supplier

Es bastante frecuente encontrar que estamos desarrollando un sistema que depende de otros sistemas para hacer su trabajo, como por ejemplo puede ser un sistema de análisis o un sistema de toma de decisiones. En este tipo de sistemas suelen existir dos

contextos, en un contexto se encuentra nuestro sistema, que "consume" al sistema del que depende y que se encuentra en el otro contexto.

Las dependencias entre los dos contextos son en una sola dirección, del contexto "cliente" al contexto "proveedor", del sistema dependiente hacia el sistema dependido.

En este tipo de relaciones el cliente puede verse limitado por necesitar funcionalidades del sistema proveedor, y al mismo tiempo el contexto proveedor puede cohibirse a la hora de realizar cambios por miedo a provocar la aparición de bugs en el contexto o contextos clientes. Para solucionar este tipo de problemas la clave es la comunicación entre los equipos de los distintos contextos. Los miembros del equipo de los contextos clientes deberían participar como clientes en las reuniones de planificación del equipo proveedor para priorizar las historias de usuario del sistema proveedor, y se debería crear conjuntamente un juego de pruebas de aceptación para el sistema proveedor, de forma que quede perfectamente definida la interfaz que esperan los contextos clientes, y el contexto proveedor pueda realizar cambios sin miedo a cambiar por error la interfaz que esperan los contextos clientes.

2.8.3.- Conformista

La relación cliente/proveedor requiere de la colaboración entre los equipos de los distintos contextos. Esta situación suele ser bastante ideal, y en la mayoría de los casos el contexto proveedor tiene sus propias prioridades y no está dispuesto a atender a las necesidades del contexto cliente. En este tipo de situaciones donde nuestro contexto depende de otro contexto sobre el cual no tenemos control alguno, (no podemos realizar modificaciones ni pedir funcionalidades) y con el que tenemos una estrecha relación, (el coste de la traducción de las comunicaciones de un contexto a otro es elevado) podemos emplear un acercamiento conformista, que consiste en acomodar nuestro modelo al expuesto por el otro contexto. Esto limita nuestro modelo a hacer simples adiciones al modelo del otro contexto, y limita la forma que puede tomar nuestro modelo. No obstante no es una idea descabellada, ya que posiblemente el otro modelo incorpore el conocimiento acumulado en el desarrollo del otro contexto. La decisión de seguir una relación de conformismo depende en gran medida de la calidad del modelo del otro contexto. Si no es adecuado, debe seguirse un enfoque más defensivo como puede ser un Anti-corruption layer o Separate ways como veremos a continuación.

2.8.4.- Anti-corruption Layer

Todas las relaciones que hemos visto hasta ahora presuponen la existencia de una buena comunicación entre los equipos de los distintos contextos o un modelo de un

contexto bien diseñado que puede ser adoptado por otro. ¿Pero qué ocurre cuando un contexto está mal diseñado y no queremos que este hecho influya sobre nuestro contexto? Para este tipo de situaciones podemos implementar un anti-corruption layer, que consiste en crear una capa intermedia entre contextos que se encarga de realizar la traducción entre nuestro contexto y el contexto con el que tenemos que comunicarnos. Generalmente esta comunicación la vamos a iniciar nosotros, aunque no tiene porqué ser así.

Un anti-corruption layer se compone de tres elementos: adaptadores, traductores y fachadas. Primero se diseña una fachada que simplifica la comunicación con el otro contexto y que expone solo la funcionalidad que nuestro contexto va a utilizar. Es importante tener claro que la fachada debe definirse en términos de elementos del modelo del otro contexto, ya que si no estaríamos mezclando la traducción con el acceso al otro sistema. Frente a la fachada de sitúa un adaptador que modifica la interfaz del otro contexto para adaptarla a la interfaz que espera nuestro contexto, y que hace uso de un traductor para mapear los elementos de nuestro contexto a los que espera la fachada del otro contexto.

2.8.5.- Separate ways

La integración está sobrevalorada, y muchas veces no merece la pena el coste que conlleva. Por este motivo, dos grupos de funcionalidades que no tengan relación pueden desarrollarse en contextos distintos sin comunicación entre los mismos. Si tenemos funcionalidades que necesitan hacer uso de los dos contextos siempre podemos realizar esta orquestación a más alto nivel.

2.8.6.- Open Host

Típicamente cuando desarrollamos un sistema y decidimos realizar una separación en contextos, lo normal es crear una capa intermedia de traducción entre contextos. Cuando el número de contextos es elevado la creación de estas capas de traducción supone una carga extra de trabajo bastante importante. Cuando creamos un contexto lo normal es que éste presente una fuerte cohesión y que las funcionalidades que ofrece puedan verse como un conjunto de servicios. (No hablamos de servicios web, sino simplemente servicios).

En estas situaciones lo mejor es crear un conjunto de servicios que definan un protocolo de comunicación común para que otros contextos puedan utilizar la funcionalidad del contexto. Este servicio debe mantener la compatibilidad entre versiones, aunque puede ir aumentando las funcionalidades ofrecidas. Las funcionalidades expuestas deben ser generales, si otro contexto necesita una

funcionalidad específica se crea en una capa de traducción independiente para no contaminar el protocolo de nuestro contexto.

2.9.- Implementación de bounded contexts en .NET

Como hemos indicado al principio de la sección, los bounded context son unidades organizativas destinadas a mantener la coherencia de modelos de gran tamaño. Por este motivo un bounded context puede representar desde un área de funcionalidad del sistema, hasta un sistema externo o un grupo de componentes destinados a realizar una tarea de forma óptima. No existe por tanto una regla general para implementar un bounded context, pero en este apartado trataremos los aspectos más importantes y pondremos algunos ejemplos de situaciones típicas.

En nuestra arquitectura dividimos el dominio y las funcionalidades en módulos de gran tamaño. Cada módulo es lógico que vaya asignado a un grupo de trabajo distinto, y que presente un conjunto de funcionalidades muy cohesivas, que pueden exponerse como un conjunto de servicios. Lo más lógico cuando tenemos que tratar con varios módulos es utilizar una relación “separate ways” entre módulos. Cada módulo a su vez será un “open host” que ofrecerá al resto un conjunto de funcionalidades en forma de servicios. De esta forma, cualquier funcionalidad que implique varios módulos se orquestará desde un nivel superior. Cada módulo se encargará por tanto de su propio modelo de objetos, y de la gestión de la persistencia del mismo. En caso de utilizar Entity Framework esto significa que tendremos una correspondencia de 1 a 1 entre módulo y contexto de entity framework.

Dentro de cada módulo es bastante probable que exista complejidad suficiente como para que podamos seguir partiendo el sistema en contextos más pequeños. No obstante, estos contextos de trabajo estarán más relacionados y presentarán una relación de comunicación basada en un “shared kernel” o “customer/suplier”. En estos casos, el contexto es más una unidad organizativa que funcional. Los distintos contextos compartirán el mismo modelo de entity framework, pero la modificación de ciertos objetos clave estará sujeta al acuerdo entre los dos equipos de los distintos contextos.

Por último, queda tratar el aspecto específico de la relación de nuestro sistema con sistemas externos o componentes de terceros, que claramente son bounded context distintos. Aquí el enfoque puede ser aceptar el modelo del sistema externo, adoptando una postura “conformista” o podemos proteger nuestro dominio mediante un “anti-corruption layer” que traduzca nuestros conceptos a los conceptos del otro contexto. La decisión entre seguir un enfoque conformista u optar por un anti-corruption layer depende de la calidad del modelo del otro contexto y del coste de la traducción de nuestro contexto al otro contexto.

2.9.1.- ¿Cómo partir un modelo de Entity Framework?

El primer paso para partir un modelo es identificar los puntos donde existen entidades con poca relación entre ellas. No es imprescindible que no exista relación alguna entre entidades, y ahora veremos por qué. Examinemos en detalle una relación. ¿Cuál es la utilidad de que exista una relación entre dos entidades? Típicamente que una de las entidades hace uso de funcionalidad de la otra para implementar su propia funcionalidad. Como por ejemplo puede ser una entidad cuenta y una entidad cliente, en la que el patrimonio de un cliente se calcula a través de la agregación del balance de todas sus cuentas y propiedades.

De forma genérica, una relación entre dos entidades puede sustituirse por una consulta en el repositorio de una de las dos entidades. Esta consulta representa la asociación. En los métodos de la otra entidad podemos añadir un parámetro extra que contiene la información de la asociación como el resultado de la consulta al repositorio y puede operar de la misma forma que si la relación existiese.

La interacción entre las dos entidades se orquestará a nivel de servicio, ya que este tipo de interacción no es muy común y la lógica no suele ser compleja. En caso de que haya que modificar una asociación (por ejemplo añadiendo o eliminando algún elemento) tendremos en dichas entidades métodos de consulta que devolverán valores booleanos para indicar si dicha acción se debe llevar a cabo o no. En lugar de tener métodos para modificar la asociación que hemos eliminado. Siguiendo con nuestro ejemplo de las cuentas y los clientes, imaginemos que queremos calcular los intereses a pagar a un determinado cliente, que variarán dependiendo de las características del cliente. Este servicio además debe guardar los intereses en una nueva cuenta si exceden en una determinada cantidad dependiendo de la antigüedad del cliente. (Ya sabemos que en realidad no se hace así, pero es solo un caso ilustrativo) Tendríamos un servicio con la siguiente interfaz:

```
public  interface IInterestRatingService
{
    void RateInterests(int clientId);
}
```

```
public class InterestRatingService : IInterestRatingService
{
   public InterestRatingService(IClientService clients,
                                 IBankAccountService accounts)
   {
     …
   }
   public void RateInterests(int clientId)
   {
       Client client = _clients.GetById(clientId);
       IEnumerable<BankAccount> clientAccounts =
              accounts.GetByClientId(clientId);
```

```
            double interests = 0;
            foreach(var account in clientAccounts)
            {
                interests += account.calculateRate(client);
            }
            if(client.ShouldPlaceInterestsInaNewAccount(interests))
            {
                BankAccount newAccount = new Account(interests);
                accounts.Add(newAccount);
            }else{
                clientAccounts.First().Charge(interests);
            }
        }

    }
```

2.9.2.- Relación entre bounded contexts y ensamblados

La existencia de un bounded context no implica directamente la creación de un ensamblado específico para él, sino que dependiendo de las relaciones del mapa de contextos, unos bounded contexts irán en el mismo ensamblado mientras que otros estarán separados. Lo normal, es que cuando dos bounded contexts tienen una relación fuerte, como por ejemplo la determinada por un shared kernel o un customer/supplier, dichos contextos se encuentren dentro del mismo ensamblado. En relaciones más débiles como pueden ser la interacción entre módulos, existen dos aproximaciones.

Una aproximación es tener todos los módulos en un mismo ensamblado, y utilizar solo los ensamblados para la división por capas. De esta manera se facilita la interacción entre módulos, al poder estos albergar referencias a elementos de cualquier otro módulo. Además tenemos la ventaja de tener todo nuestro dominio en un solo ensamblado, lo que simplifica el despliegue y la reutilización del dominio en otras aplicaciones. Hay que destacar que el hecho de que todos los módulos estén en el mismo ensamblado no significa que compartan el mismo contexto de Entity Framework. Este es el enfoque que hemos seguido en el ejemplo de interacción entre módulos.

La otra aproximación es tener cada módulo en un ensamblado distinto. Con esto no solo mejoramos sino que garantizamos el aislamiento entre módulos, pero las comunicaciones entre módulos se vuelven un poco complicadas. Cada módulo debería definir sus propias abstracciones de las entidades de otro módulo que necesite (que debieran reducirse al máximo), y en un nivel superior, a través de un anticorruption layer, crear un adaptador de las entidades del otro módulo a las abstracciones definidas en el módulo.

2.10.- Visión de tecnologías en Arquitectura N-Layer

Antes de entrar en los detalles de cómo definir la estructura de la solución de Visual Studio, conviene tener una visión de alto nivel donde estén relacionadas/mapeadas las diferentes capas anteriormente descritas con sus tecnologías respectivas:

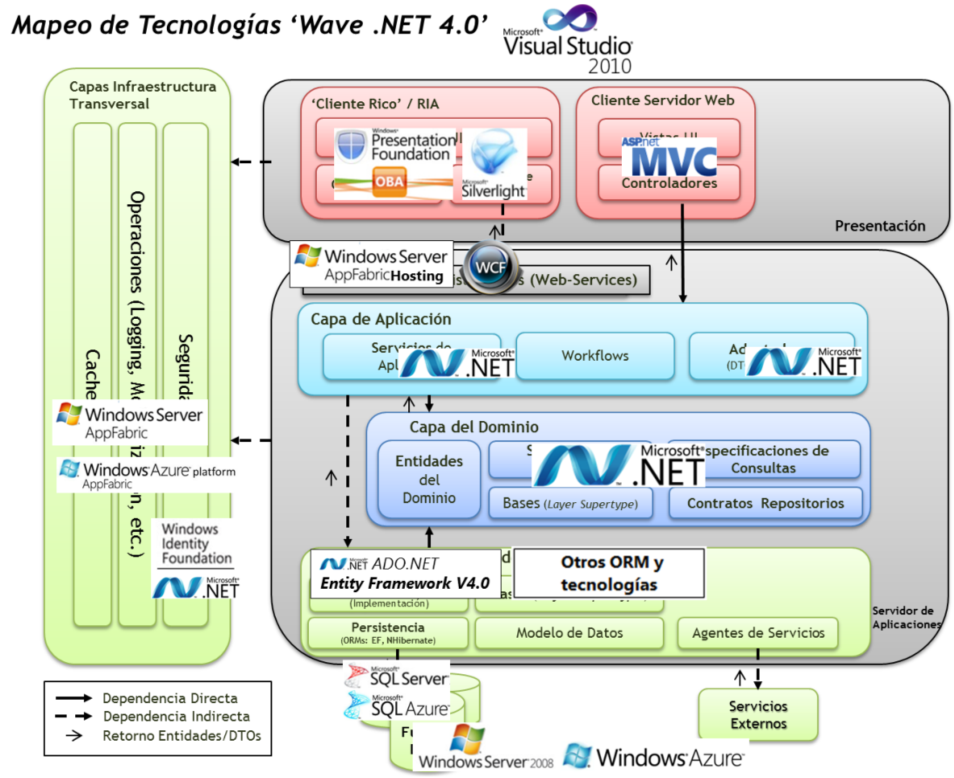


Figura 10.- Mapeo de tecnologías 'OLA.NET 4.0'

En los siguientes capítulos iremos entrando en detalle sobre cómo implementar los diferentes patrones de la arquitectura con cada una de las tecnologías situadas en el gráfico.

2.11.-Implementación de Estructura de Capas en Visual Studio 2010

Para implementar una Arquitectura en Capas (según nuestro modelo lógico definido, orientado a Arquitecturas N-Capas DDD), hay una serie de pasos que debemos realizar:

1.- La solución de Visual Studio debe estar organizada y mostrar de forma clara y obvia donde está situada la implementación de cada capa y sub-capa.

2.- Cada capa tiene que estar correctamente diseñada y necesita incluir los patrones de diseño y tecnologías de cada capa.

3.- Existirán capas transversales de patrones y tecnologías a ser utilizados a lo largo de toda la aplicación, como la implementación de la tecnología escogida para IoC, o aspectos de seguridad, etc. Estas capas transversales (Infraestructura Transversal de DDD) serán capas bastante reutilizables en diferentes proyectos que se realicen en el futuro. Es un mini-framework, o mejor llamado *seedwork*, en definitiva, un código fuente que será reutilizado también en otros proyectos futuros, así como ciertas clases base (*Core*) de las capas del Dominio y Persistencia de Datos.

2.12.-Aplicación ejemplo N-Layer DDD con .NET 4.0

Prácticamente todos los ejemplos de código y estructuras de proyecto/*solutions* que se muestran en la presente guía pertenecen a la aplicación ejemplo que se ha desarrollado para acompañar este libro. Recomendamos encarecidamente bajar el código fuente de Internet e irlo revisando según se explica en el libro, pues siempre se podrán observar más detalles directamente en el código real.
La aplicación ejemplo está publicada en CODEPLEX, con licencia OPEN SOURCE, en esta URL:

http://microsoftnlayerapp.codeplex.com/

2.13.-Diseño de la solución de Visual Studio

Teniendo una 'Solución' de Visual Studio, inicialmente crearemos la estructura de carpetas lógicas para albergar y distribuir los diferentes proyectos. En la mayoría de los casos crearemos un proyecto (.DLL) por cada capa o sub-capa, para disponer así de una mayor flexibilidad y facilitar los posibles desacoplamientos. Sin embargo, esto ocasiona un número de proyectos considerable, por lo que es realmente imprescindible ordenarlos/jerarquizarlos mediante carpetas lógicas de Visual Studio.

La jerarquía inicial sería algo similar a la siguiente:

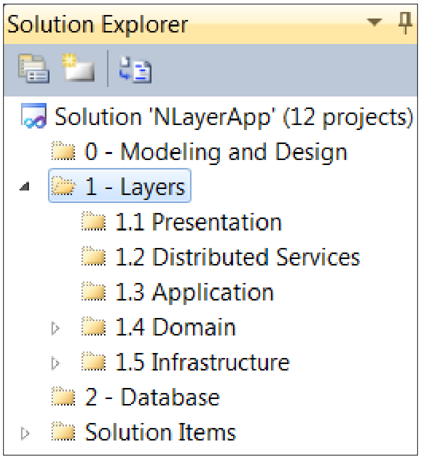


Figura 11.- Jerarquía de Carpetas en Solución de Visual Studio

Empezando por arriba, la primera carpeta ('*0 – Modeling & Design*') contendrá los diferentes diagramas de Arquitectura y Diseño realizados con VS.2010, como diagrama Layer de la Arquitectura, y diferentes diagramas UML de diseño interno. Estos diagramas los iremos utilizando para representar la implementación que hagamos.

La numeración de las capas es simplemente para que aparezcan en un orden adecuado siguiendo el orden real de la arquitectura y sea más sencillo buscar cada capa dentro del *solution* de Visual Studio.

La siguiente carpeta '1 Layers', contendrá las diferentes capas de la Arquitectura N-Layer, como se observa en la jerarquía anterior.

Capa de Presentación

La primera capa, Presentación, contendrá los diferentes tipos de proyectos que pudiera haber, es decir, proyectos Windows-*Rich* (WPF, WinForms, OBA), RIA (Silverlight), Web (ASP.NET) o Windows Phone, etc.:

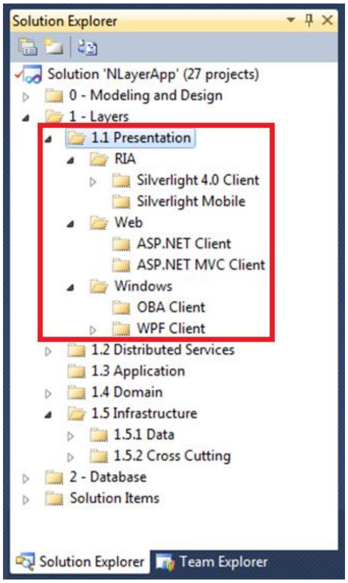


Figura 12.- Capas de Presentación

Posteriormente, tenemos las capas de componentes que normalmente están situadas en un servidor de aplicaciones (aunque ahí estaríamos hablando de despliegue, y eso puede variar, por lo que a nivel de organización en VS, no especificamos detalles de despliegue). En definitiva, dispondremos de las diferentes Capas principales de una Arquitectura N-*Layered* Orientada al Dominio, con diferentes proyectos para cada subcapa:

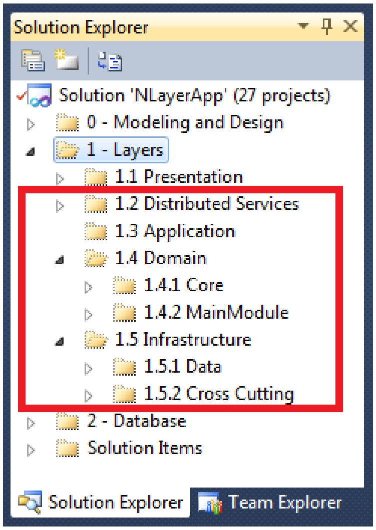


Figura 13.- Capas del Servidor de Aplicaciones

Dentro de cada una de dichas carpetas, añadiremos los proyectos necesarios según los elementos típicos de cada capa. Esto también viene determinado dependiendo de los patrones a implementar (explicados posteriormente a nivel lógico e implementación en la presente guía).

Capa de Servicios Distribuidos (Servicios WCF)

Esta Capa es donde implementaremos los Servicios WCF (normalmente Servicios-Web) para poder acceder remotamente a los componentes del Servidor de aplicaciones. Es importante destacar que esta capa de Servicios Distribuidos es opcional, puesto que en algunos casos (como capa de presentación web ASP.NET), es posible que se acceda directamente a los componentes de APPLICATION y DOMAIN, si el servidor Web de ASP.NET está en el mismo nivel de servidores que los componentes de negocio.

En el caso de hacer uso de servicios distribuidos para accesos remotos, la estructura puede ser algo similar a la siguiente:

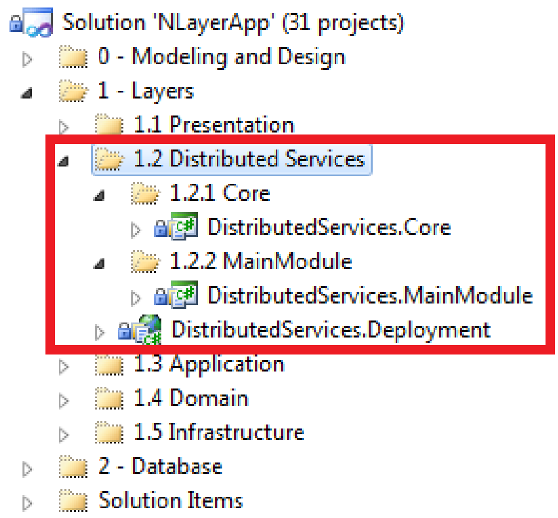


Figura 14.- Uso de servicios distribuidos

Un proyecto para el *Hoster* del Servicio WCF, es decir, el proceso donde se ejecuta y publica el servicio WCF. Ese proyecto/proceso puede ser de tipo *WebSite* de IIS (o *Casini* para desarrollo), un Servicio Windows, o realmente, cualquier tipo de proceso.

Y donde realmente está la funcionalidad del Servicio-Web es en los Servicios que exponen la lógica de cada módulo, es decir, dispondremos de **un proyecto de Servicio WCF (.DLL) por cada MODULO funcional de la aplicación**. En nuestro ejemplo, tenemos solo un módulo llamado '**MainModule**'.

En el caso de *hosting* de Servidor Web, internamente se añadirá un .SVC por cada MÓDULO de la aplicación.

Adicionalmente, deberá haber también un proyecto de clases de *Testing* (Pruebas Unitarias), dentro de esta capa.

Para un Servicio WCF en producción, se recomienda que el proyecto sea de tipo WebSite desplegado en IIS (**IIS 7.x**, a ser posible, para tener como posibilidad el utilizar *bindings* como *NetTCP* y no solamente *bindings* basados en HTTP), e incluso en el mejor escenario de despliegue, con IIS más ***Windows Server AppFabric*** para disponer de la monitorización e instrumentalización de los Servicios WCF, proporcionado por ***AppFabric***.

Capa de Aplicación

Como se ha explicado anteriormente en la parte de Arquitectura lógica de esta guía, esta capa no debe contener realmente reglas del dominio o conocimiento de la lógica de negocio, simplemente debe realizar tareas de coordinación de aspectos tecnológicos de la aplicación que nunca explicaríamos a un experto del dominio o usuario de negocio. Aquí implementamos la coordinación de la 'fontanería' de la aplicación, como coordinación de transacciones, ejecución de unidades de trabajo, uso mayoritario de Repositorios y llamadas a objetos del Dominio.

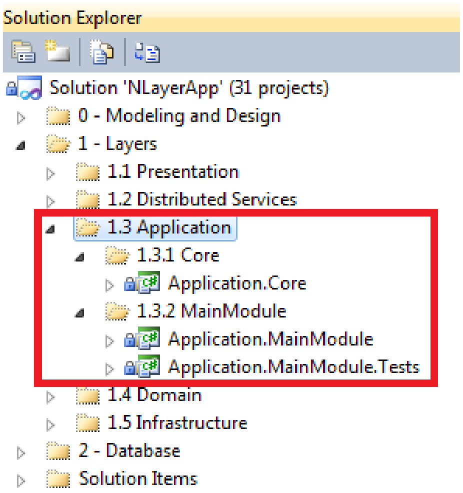


Figura 15.- Sub-Capas de Aplicación

Cada capa con clases lógicas tendrá a su vez un proyecto de clases de *Testing* (Pruebas Unitarias).

Capa de Dominio

Esta es la Capa más importante desde el punto de vista de la problemática de la aplicación, puesto que es aquí donde implementamos toda la lógica del dominio, entidades del dominio, etc.

Esta Capa tiene internamente varias sub-capas o tipos de elementos. Se recomienda consolidar al máximo el número de proyectos requerido dentro de una misma capa. Sin embargo, en este caso, es bueno disponer de un ensamblado/proyecto específico para las entidades, para que no estén acopladas a los Servicios del Dominio:

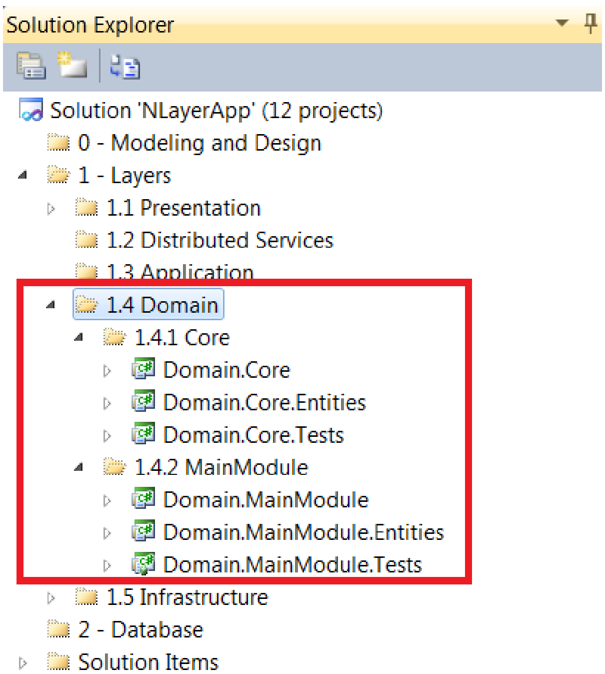


Figura 16.- Sub-Capas del Dominio

A nivel general, podemos disponer de un proyecto '*Core*' de clases base y otras clases reutilizables de forma horizontal en todos los módulos funcionales del Dominio.

Por cada MODULO funcional de la aplicación (en el ejemplo, en este caso el llamado '*MainModule*'), implementaremos toda la lógica del módulo (Servicios, Especificaciones y Contratos de Repositorios) dentro de un único proyecto (en este caso Domain.MainModule), pero necesitamos un proyecto aislado para las '**Entidades del Dominio**', por cada MÓDULO, donde Entity-Framework nos genere nuestras clases entidad POCO o Self-Tracking.

Este es el contenido de los proyectos de *Domain* a nivel de un módulo:

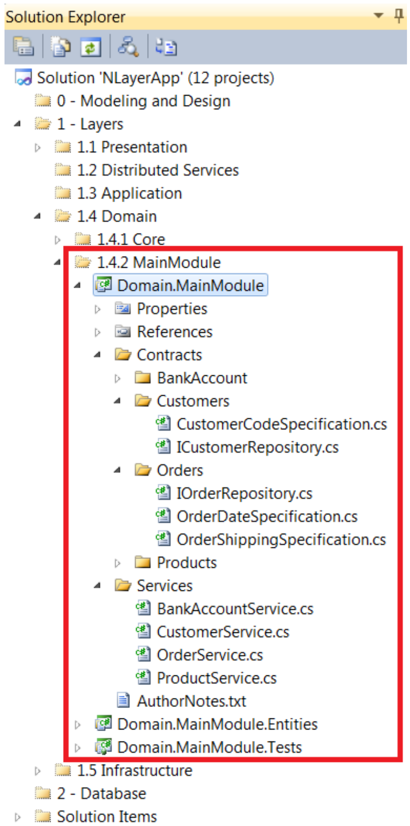


Figura 17.- Contenido de los proyectos de Dominio

Cada proyecto con clases lógicas tendrá a su vez un proyecto de clases de *Testing* (Pruebas Unitarias) y pudiéramos tener otros proyectos de pruebas de integración y funcionales.

Esta capa de Dominio se explica tanto a nivel lógico como de implementación en un capítulo completo de la guía.

Capa de Infraestructura de Persistencia de Datos

La parte más característica de esta capa es la implementación de REPOSITORIOS para realizar la persistencia y acceso a datos. En este módulo es también donde por lo tanto implementamos todo lo relacionado con el modelo y conexiones/acciones a la base de datos de ENTITY FRAMEWORK.

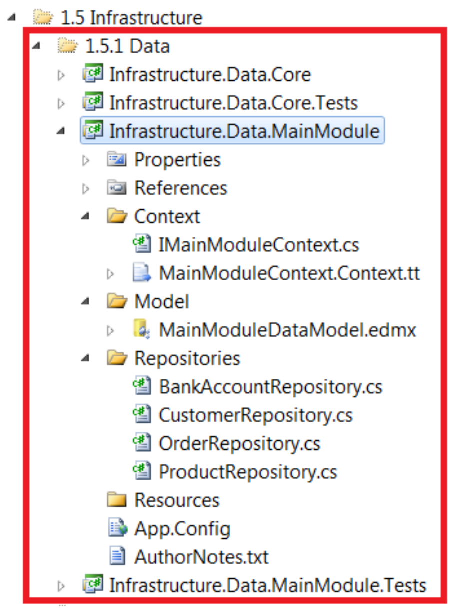


Figura 18.- Capa de Infraestructura de Persistencia de Datos

A nivel de cada MODULO funcional (en este caso, *MainModule*) dispondremos de de un único proyecto con los siguientes elementos:

- **'DataModel'**: Contendrá el modelo de *EntityFramework*. Si bien, las clases que genera Entity Framework (*Container* y Entidades POCO/IPOCO) las extraeremos a otros proyectos para poder desacoplarlo según el diseño del Dominio en DDD. Aquí solo estará el modelo de datos (En nuestro caso, *MainModuleDataModel.edmx*).

- **'Context'** implementa una abstracción del contexto/contenedor de EntityFramework, para poder sustituirlo por un *fake/mock* y realizar pruebas unitarias.

- **Repositorios** (Repositories): Clases encargadas de la lógica de persistencia de datos.

También dispondremos de otro proyecto para los Tests de todo el módulo.

Los proyectos de tipo '**Core**' son proyectos a utilizar para implementar clases base y extensiones que son válidos para reutilizar de forma horizontal en la implementación de Capa de Persistencia de todos los módulos funcionales de la aplicación.

Esta capa de 'Persistencia de Datos' se explica tanto a nivel lógico como de implementación en un capítulo completo de la guía.

2.14.- Arquitectura de la Aplicación con Diagrama Layer de VS.2010

Para poder diseñar y comprender mejor la Arquitectura, en VS.2010 disponemos de un diagrama donde podemos plasmar la Arquitectura N-Layered que hayamos diseñado y adicionalmente nos permite mapear las capas que dibujemos visualmente con *namespaces* lógicos y/o *assemblies* de la solución. Esto posibilita posteriormente el validar la arquitectura con el código fuente real, de forma que se compruebe si se están realizando accesos/dependencias entre capas no permitidas por la arquitectura, e incluso ligar estas validaciones al proceso de control de código fuente en TFS.

En nuestro ejemplo de aplicación, este sería el diagrama de **Arquitectura N-Layer**:

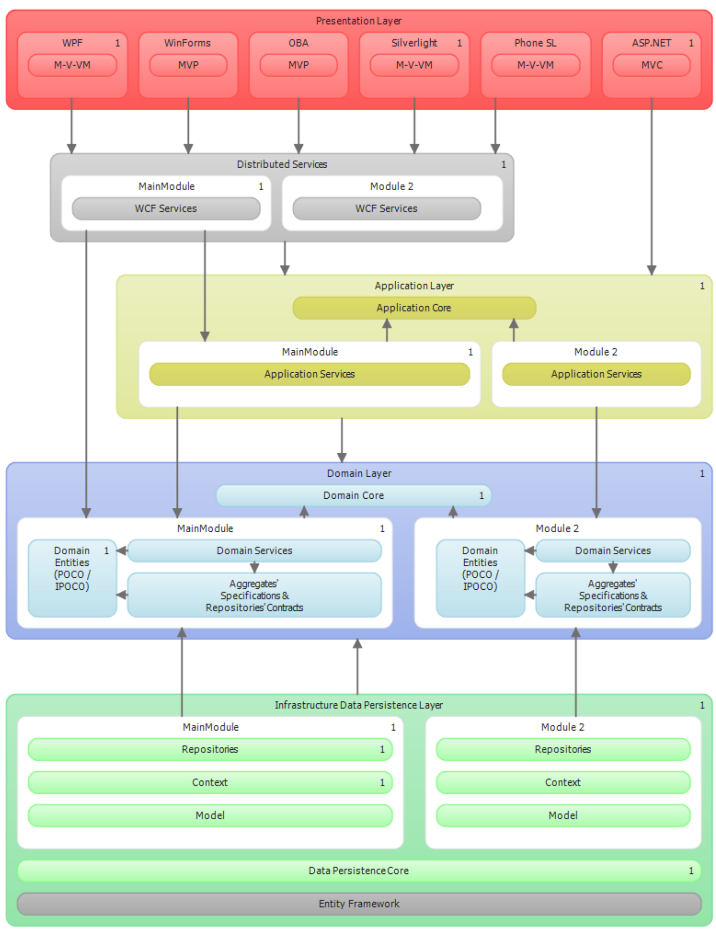


Figura 19.- Arquitectura N-Layer DDD en VS.2010

En los siguientes capítulos de la guía analizamos la lógica e implementación de cada una de estas capas y sub-capas. Sin embargo, resaltamos algunos aspectos globales a continuación.

Como se puede observar en el diagrama de Arquitectura, la Capa Central sobre la que gira toda la Arquitectura, es la Capa de Dominio. Esto es también apreciable a nivel de las dependencias. La mayoría de las dependencias finalizan en la Capa de Dominio (p.e. dependencias con las Entidades del Dominio, etc.). Y la Capa de Dominio, a su vez tiene mínimas dependencias con otras capas (Infraestructura, Persistencia de Datos), y en esos casos, son 'dependencias desacopladas', es decir, basadas en abstracciones (interfaces) y a través de contenedores de IoC, por lo que no aparecen esas dependencias de forma directa como 'flechas' en el diagrama.

Cabe destacar, que por claridad en el diagrama anterior, no se han especificado todas las dependencias reales de más bajo nivel que tiene la aplicación, ejemplo de la cual se ha obtenido este diagrama de Capas.

Otro aspecto a mencionar es que la Capa de 'Servicios Remotos' o 'Servicios Distribuidos' (Servicios WCF, en definitiva, en .NET), es una capa opcional dependiendo del tipo de Capa de Presentación a utilizar. Si la capa de presentación se ejecuta en un entorno remoto (Silverlight, WPF, Winforms, OBA, se ejecutan en el ordenador cliente), está claro que será necesario. Pero por ejemplo, en el caso de un cliente Web (ASP.NET o ASP.NET MVC), cabe la posibilidad más normal de que el servidor web de capa de presentación esté en el mismo nivel físico de servidores que los componentes de negocio. En ese caso, no tiene sentido hacer uso de servicios WCF, puesto que impactaría innecesariamente en el rendimiento de la aplicación.

En cuanto a la 'Capa de Aplicación', va a ser normalmente nuestra capa 'Fachada', donde se exponen los Servicios de Aplicación que coordinan las tareas y acciones a efectuar contra el Dominio así como contra la persistencia y consulta de datos.

2.15.- Implementación de Inyección de Dependencias e IoC con UNITY

En la presente sección se pretende explicar las técnicas y tecnologías para realizar una implementación específica del desacoplamiento entre capas de la Arquitectura. En concreto, explicar las técnicas **DI (Inyección de dependencias)** e **IoC (Inversión de Control)** con una tecnología concreta de **Microsoft Pattern & Practices**, llamada **Unity**.

DI e **IoC** se pueden implementar con diversas tecnologías y frameworks de diferentes fabricantes, como:

Tabla 8.- Implementaciones de Contenedores IoC

Framework	Implementador	Información
Unity http://msdn.microsoft.com/en-us/library/dd203101.aspx http://unity.codeplex.com/	Microsoft Pattern & Practices	Es actualmente el *framework* ligero de Microsoft más completo para implementar IoC y DI. Es un proyecto *OpenSource*. Con licenciamiento de tipo Microsoft Public License (Ms-PL)
Castle Project (Castle Windsor) http://www.castleproject.org/	CastleStronghold	Castle es un proyecto OpenSource. Es uno de los mejores frameworks para IoC y DI.
MEF (Microsoft Extensibility Framework) http://code.msdn.microsoft.com/mef http://www.codeplex.com/MEF	Microsoft (Forma parte de .NET 4.0)	Es actualmente un *framework* para extensibilidad automática de herramientas y aplicaciones, no está tan orientado a desacoplamiento entre Capas de Arquitectura utilizando IoC y DI.
Spring.NET http://www.springframework.net/	SpringSource	Spring.NET es un proyecto OpenSource. Es uno de los mejores frameworks con AOP (Aspect Oriented Programming), ofreciendo también capacidades IoC.

StructureMap http://structuremap.sourceforge.net/Default.htm	Varios desarrolladores de la comunidad .NET	Proyecto OpenSource.
Autofac http://code.google.com/p/autofac/	Varios desarrolladores de la comunidad .NET	Proyecto OpenSource.
LinFu http://code.google.com/p/linfu/downloads/list http://www.codeproject.com/KB/cs/LinFuPart1.aspx	Varios desarrolladores de la comunidad .NET	Proyecto OpenSource. Aporta IoC, AOP y otras características.

Para el ejemplo de aplicación N-Capas de nuestra Arquitectura marco, hemos escogido UNITY por ser actualmente el *framework* IoC y DI más completo ofrecido por Microsoft. Pero por supuesto, en una arquitectura marco empresarial, podría hacerse uso de cualquier *framework* IoC (listado o no en la tabla anterior).

2.15.1.- Introducción a Unity

El *Application Block* denominado **Unity** (implementado por *Microsoft Patterns & Practices*), es un contenedor de inyección de dependencias extensible y ligero (Unity no es un gran framework pesado). Soporta inyección en el constructor, inyección de propiedades, inyección en llamadas a métodos y contenedores anidados.

Básicamente, *Unity* es un contenedor donde podemos registrar tipos (clases, interfaces) y también mapeos entre dichos tipos (como un mapeo de un interfaz hacia una clase) y además el contenedor de *Unity* puede instanciar bajo demanda los tipos concretos requeridos.

Unity está disponible como un *download* público desde el site de Microsoft (es gratuito) y también está incluido en la Enterprise Library 4.0/5.0 y en PRISM *(Composite Applications Framework)*, los cuales hacen uso extensivo de *Unity*.

Para hacer uso de *Unity*, normalmente registramos tipos y mapeos en un contenedor de forma que especificamos las dependencias entre interfaces, clases base y tipos concretos de objetos. Podemos definir estos registros y mapeos directamente por código fuente o bien, como normalmente se hará en una aplicación real, mediante

XML de ficheros de configuración. También se puede especificar inyección de objetos en nuestras propias clases haciendo uso de atributos que indican las propiedades y métodos que requieren inyección de objetos dependientes, así como los objetos especificados en los parámetros del constructor de una clase, que se inyectan automáticamente.

Incluso, se puede hacer uso de las extensiones del contenedor que soportan otras cosas como la extensión "*Event Broker*" que implementa un mecanismo de publicación/suscripción basado en atributos, que podemos utilizar en nuestras aplicaciones. Podríamos incluso llegar a construir nuestras propias extensiones de contenedor.

Unity proporciona las siguientes ventajas al desarrollo de aplicaciones:

- Soporta abstracción de requerimientos; esto permite a los desarrolladores el especificar dependencias en tiempo de ejecución o en configuración y simplifica la gestión de aspectos horizontales (*crosscutting concerns*), como puede ser el realizar pruebas unitarias contra *mocks* y *stubs*, o contra los objetos reales de la aplicación.

- Proporciona una creación de objetos simplificada, especialmente con estructuras de objetos jerárquicos con dependencias, lo cual simplifica el código de la aplicación.

- Aumenta la flexibilidad al trasladar la configuración de los componentes al contenedor IoC.

- Proporciona una capacidad de localización de servicios; esto permite a los clientes el guardar o cachear el contenedor. Es por ejemplo especialmente útil en aplicaciones web ASP.NET donde los desarrolladores pueden persistir el contenedor en la sesión o aplicación ASP.NET.

2.15.2.- Escenarios usuales con Unity

Unity resuelve problemas de desarrollo típicos en aplicaciones basadas en componentes. Las aplicaciones de negocio modernas están compuestas por objetos de negocio y componentes que realizan tareas específicas o tareas genéricas dentro de la aplicación, además solemos tener componentes que se encargan de aspectos horizontales de la arquitectura de la aplicación, como pueden ser trazas, logging, autenticación, autorización, cache y gestión de excepciones.

La clave para construir satisfactoriamente dichas aplicaciones de negocio (aplicaciones N-Capas), es conseguir un diseño desacoplado *(decoupled / very loosely coupled).* Las aplicaciones desacopladas son más flexibles y fácilmente mantenibles y especialmente son más fáciles de probar durante el desarrollo (Pruebas Unitarias). Se pueden realizar **mocks** (simulaciones) de objetos que tengan fuertes

dependencias concretas, como conexiones a bases de datos, conexiones a red, conexiones a aplicaciones externas como ERPs, etc. De forma que las pruebas unitarias se puedan realizar contra los mocks o contra los objetos reales cambiándolo de una forma dinámica o basado en configuración.

La inyección de dependencias es una técnica fundamental para construir aplicaciones desacopladas. Proporciona formas de gestionar dependencias entre objetos. Por ejemplo, un objeto que procesa información de un cliente puede depender de otros objetos que acceden a la base de datos, validan la información y comprueban que el usuario está autorizado para realizar actualizaciones. Las técnicas de inyección de dependencias pueden asegurar que la clase 'Cliente' instancie y ejecute correctamente dichos objetos de los que depende, especialmente cuando las dependencias son abstractas.

2.15.3.- Patrones Principales

Los siguientes patrones de diseño definen aproximaciones de arquitectura y desarrollo que simplifican el proceso:

Patrón de Inversión de Control (IoC). Este patrón genérico describe técnicas para soportar una arquitectura tipo 'plug-in' donde los objetos pueden buscar instancias de otros objetos que requieren.

Patrón de Inyección de Dependencias (DI). Es realmente un caso especial de IoC. Es una interfaz de técnica de programación basada en alterar el comportamiento de una clase sin cambiar el código interno de la misma. Los desarrolladores codifican contra un interfaz relacionado con la clase y usan un contenedor que inyecta instancias de objetos dependientes en la clase basada en el interfaz o tipo de objeto. Las técnicas de inyección de instancias de objetos son 'inyección de interfaz', 'inyección de constructor', inyección de propiedad (*setter*), e inyección de llamada a método.

Patrón de Intercepción. Este patrón introduce otro nivel de indirección. Esta técnica sitúa un objeto entre el cliente y el objeto real. Se utiliza un proxy entre el cliente y el objeto real. El comportamiento del cliente es el mismo que si interactuara directamente con el objeto real, pero el proxy lo intercepta y resuelve su ejecución colaborando con el objeto real y otros objetos según requiera.

2.15.4.- Métodos principales

Unity expone dos métodos para registrar tipos y mapeos en el contenedor:

RegisterType(): Este método registra un tipo en el contendor. En el momento adecuado, el contenedor construye una instancia del tipo especificado. Esto puede ser en respuesta a una inyección de dependencias iniciada mediante atributos de clase o cuando se llama al método Resolve. El tiempo de vida (lifetime) del objeto corresponde

al tiempo de vida que se especifique en los parámetros del método. Si no se especifica valor al *lifetime*, el tipo se registra de forma transitoria, lo que significa que el contenedor crea una nueva instancia en cada llamada al método **Resolve ().**

RegisterInstance(): Este método registra en el contendor una instancia existente del tipo especificado, con un tiempo de vida especificado. El contenedor devuelve la instancia existente durante ese tiempo de vida. Si no se especifica un valor para el tiempo de vida, la instancia tiene un tiempo de vida controlada por el contenedor.

2.15.5.- Registro Configurado de tipos en Contenedor

Como ejemplo de uso de los métodos **RegisterType** y **Resolve**, a continuación realizamos un registro de un mapeo de un interfaz llamado **ICustomerService** y especificamos que el contenedor debe devolver una instancia de la clase **CustomerService** (la cual tendrá implementado el interfaz **ICustomerService**).

```
C#
//Registro de tipos en Contenedor de UNITY
IUnityContainer container = new UnityContainer();
container.RegisterType<ICustomerManagementService,
CustomerManagementService>();
...
...
//Resolución de tipo a partir de Interfaz
ICustomerManagementService customerSrv =
container.Resolve<IICustomerManagementService>();
```

Normalmente en la versión final de aplicación, el registro de clases, interfaces y mapeos en el contenedor, se puede realizar de forma declarativa en el XML de los ficheros de configuración, quedando completamente desacoplado. Sin embargo, tal y como se muestra en las líneas de código anteriores, durante el desarrollo probamente es más cómodo realizarlo de forma '*Hard-coded*', pues así los errores tipográficos se detectarán en tiempo de compilación en lugar de en tiempo de ejecución (como pasa con el XML).

Con respecto al código anterior, la línea que siempre estará en el código de la aplicación, sería la que instancia propiamente el objeto resolviendo la clase que debe utilizarse mediante el contenedor, es decir, la llamada al método **Resolve()** (Independientemente de si se realiza el registro de tipos por XML o '*Hard-Coded*').

2.15.6.- Inyección de dependencias en el constructor

Como ejemplo de inyección en el constructor, si instanciamos una clase usando el método Resolve() del contenedor de Unity, y dicha clase tiene un constructor con uno o

más parámetros (dependencias con otras clases), el contenedor de Unity creará automáticamente las instancias de los objetos dependientes especificados en el constructor.

A modo de ejemplo, tenemos un código inicial que no hace uso de Inyección de Dependencias ni tampoco de Unity. Queremos cambiar esta implementación para que quede desacoplado, utilizando IoC mediante Unity. Tenemos en principio un código que utiliza una clase de negocio llamada **CustomerManagementService.** Es una simple instanciación y uso:

```
C#
…
{
    CustomerManagementService custService =
                                  new CustomerManagementService ();
    custService.SaveData("0001", "Microsoft", "Madrid");
}
```

Este código es importante tener en cuenta que sería el código a implementar en el inicio de una acción, por ejemplo, sería el código que implementaríamos en un método de un Servicio-Web WCF.

A continuación tenemos la definición de dicha **clase de Servicio inicial sin inyección de dependencias (CustomerManagementService),** que hace uso a su vez de una clase de la capa de acceso a datos, llamada **CustomerRepository** (clase repositorio o de acceso a datos):

```
C#

public class CustomerManagementService
{
    //Members
    private ICustomerRepository _custRepository;
    //Constructor
    public CustomerManagementService()
    {
        _custRepository = new CustomerRepository();
    }
    public SaveData()
    {
        _custRepository.SaveData("0001", "Microsoft", "Madrid");
    }
}
```

Hasta ahora, en el código anterior, no tenemos nada de IoC ni DI, no hay inyección de dependencias ni uso de Unity, es todo código tradicional orientado a objetos. Ahora vamos a modificar la clase de negocio CustomerManagementService de forma que la creación de la clase de la que dependemos (CustomerRepository) no lo hagamos nosotros, sino que la instanciación de dicho objeto sea hecha automáticamente por el contenedor de Unity. Es decir, tendremos un código haciendo uso de inyección de dependencias en el constructor.

```
C#

   public class CustomerManagementService
   {
       //Members
       private ICustomerRepository _custRepository;
       //Constructor
       public CustomerManagementService (ICustomerRepository
                                                 customerRepository)
       {
           _custRepository = customerRepository;
       }
       public SaveData()
       {
           _custRepository.SaveData("0001", "Microsoft", "Madrid");
       }
   }
```

Es importante destacar que, como se puede observar, no hemos hecho ningún 'new' explícito de la clase CustomerRepository. Es el contenedor de Unity el que automáticamente creará el objeto de CustomerRepository y nos lo proporcionará como parámetro de entrada a nuestro constructor. **Esa es precisamente la inyección de dependencias en el constructor.**

En tiempo de ejecución, el código de instanciación de **CustomerManagementService** se realizaría utilizando el método Resolve() del contenedor de Unity, el cual origina la instanciación generada por el *framework* de Unity de la clase CustomerRepository dentro del ámbito de la clase **CustomerManagementService**.

El siguiente código es el que implementaríamos en la capa de primer nivel que consumiría objetos del Dominio. Es decir, sería probablemente la capa de Servicios Distribuidos (WCF) o incluso capa de presentación web ejecutándose en el mismo servidor de aplicaciones (ASP.NET):

```
C# (En Capa de Servicio WCF ó Capa de Aplicación o en aplicación
ASP.NET)
…
{
    IUnityContainer container = new UnityContainer;
    CustomerManagementService custService =
                        container.Resolve<ICustomerManagementService>();
    custService.SaveData("0001", "Microsoft", "Madrid");
}
```

Como se puede observar en el uso de **Resolve<>(),** en ningún momento hemos creado nosotros una instancia de la clase de la que dependemos (CustomerRepository) y por lo tanto nosotros no hemos pasado explícitamente un objeto de CustomerRepository al constructor de nuestra clase CustomerManagementService. Y sin embargo, cuando se instancie la clase de servicio (CustomerManagementService), automáticamente se nos habrá proporcionado en el constructor una instancia nueva de CustomerRepository. Eso lo habrá hecho precisamente el contenedor de Unity al detectar la dependencia. Esta es la inyección de dependencias, y nos proporciona la

flexibilidad de poder cambiar la dependencia en tiempo de configuración y/o ejecución. Por ejemplo, si en el fichero de configuración hemos especificado que se creen objetos Mock (simulación) en lugar de objetos reales de acceso a datos (Repository), la instanciación de la clase podría haber sido la de **CustomerMockRepository** en lugar de **CustomerRepository** (ambas implementarían el mismo interfaz **ICustomerRepository**).

2.15.7.- Inyección de Propiedades (Property Setter)

A continuación mostramos la inyección de propiedades. En este caso, tenemos una clase llamada **ProductService** que expone como propiedad una referencia a una instancia de otra clase llamada Supplier (clase entidad/datos, no definida en nuestro código). Para forzar la inyección de dependencias del objeto dependiente, se debe aplicar el atributo "Dependency" a la declaración de la propiedad, como muestra el siguiente código.

C#

```
public class ProductService
{
    private Supplier supplier;
    [Dependency]
    public Supplier SupplierDetails
    {
        get { return supplier; }
        set { supplier = value; }
    }
}
```

Entonces, al crear una instancia de la clase **ProductService** mediante el contenedor de Unity, automáticamente se generará una instancia de la clase **Supplier** y se establecerá dicho objeto como el valor de la propiedad **SupplierDetails** de la clase **ProductService** .

Para más información sobre ejemplos de programación con Unity, estudiar la documentación y labs de:

Unity 1.2 Hands On Labs
http://www.microsoft.com/downloads/details.aspx?displaylang=en&FamilyID=93a5e18f-3211-44ef-b785-c59bcec4cd6f

Webcast Demos
http://unity.codeplex.com/Wiki/View.aspx?title=Webcast%20demos

MSDN Technical Article & Sample Code
http://msdn.microsoft.com/en-us/library/cc816062.aspx

2.15.8.- Resumen de características a destacar de Unity

Unity proporciona los siguientes puntos/características que merece la pena destacar:

- Unity proporciona un mecanismo para construir (o ensamblar) instancias de objetos, los cuales pueden contener otras instancias de objetos dependientes.

- Unity expone métodos “RegisterType<>()” que permiten configurar el contenedor con mapeos de tipos y objetos (incluyendo instancias singleton) y métodos “Resolve<>()” que devuelven instancias de objetos construidos que pueden contener objetos dependientes.

- Unity proporciona **inversión de control (IoC)** permitiendo inyección de objetos preconfigurados en clases construidas por el *application block*. Podemos especificar un interfaz o clase en el constructor (inyección en constructor), o podemos aplicar atributos a propiedades y métodos para iniciar inyección de propiedades e inyección de llamadas a métodos.

- Se soporta jerarquía de contenedores. Un contenedor puede tener contenedores hijo, lo cual permite que las consultas de localización de objetos pasen de los contenedores hijos a los contenedores padre.

- Se puede obtener la información de configuración de sistemas estándar de configuración, como ficheros XML, y utilizarlo para configurar el contenedor.

- No se requiere definiciones específicas en las clases. No hay requerimientos a aplicar a las clases (como atributos), excepto cuando se usa la inyección de llamada a métodos o la inyección de propiedades.

- Unity permite extender las funcionalidades de los contenedores; por ejemplo, podemos implementar métodos que permitan construcciones adicionales de objetos y características de contenedores, como cache.

2.15.9.- Cuándo utilizar Unity

La inyección de dependencias proporciona oportunidades para simplificar el código, abstraer dependencias entre objetos y generar instancias de objetos dependientes de una forma automatizada. Sin embargo, el proceso puede tener un pequeño impacto en el rendimiento (normalmente es insignificante cuando en paralelo tenemos dependencias a recursos externos como bases de datos y consumo de servicios distribuidos, que son

realmente los cuellos de botella de la mayoría de las aplicaciones). En otros casos trabajando solamente con objetos en memoria, sí que podría impactar significativamente en el rendimiento.

Así mismo, también se incrementa algo la complejidad donde simplemente existían dependencias directas.

En general:

Se debe utilizar *Unity* en las siguientes situaciones:

- Tus objetos y clases pueden tener dependencias sobre otros objetos y clases.
- Tus dependencias son complejas o requieren abstracción.
- Se quiere hacer uso de características de inyección en constructor, método o propiedad.
- Se quiere gestionar el tiempo de vida de las instancias de los objetos.
- Se quiere poder configurar y cambiar dependencias en tiempo de ejecución.
- Se quiere realizar pruebas unitarias sobre mocks/stubs.
- Se quiere poder cachear o persistir las dependencias a lo largo de *postbacks* en una aplicación Web.

No se necesita utilizar Unity en las siguientes situaciones:

- Tus objetos y clases no tienen dependencias con otros objetos y clases.
- Tus dependencias son muy simples o no requieren abstracción.

3.- ORIENTACIÓN A ARQUITECTURA EDA (EVENT DRIVEN ARCHITECTURE)

EDA (*Event-Driven Architecture*) es un patrón de arquitectura de software que promociona fundamentalmente el uso de eventos (generación, detección, consumo y reacción a eventos) como hilo conductor principal de ejecución de cierta lógica del Dominio. Es un tipo de Arquitectura genérica orientada a eventos, por lo que puede ser implementada con lenguajes de desarrollo multidisciplinar y no es necesario/obligatorio hacer uso de tecnologías especiales (si bien, las tecnologías especialmente diseñadas para implementar *Workflows* y orquestaciones de procesos de negocio, pueden ayudar mucho a esta tendencia de Arquitectura).

En la presente guía de arquitectura, EDA va a incluirse como una posibilidad complementaria, no como algo obligatorio a diseñar e implementar, pues la idoneidad de una fuerte orientación a eventos depende mucho del tipo de aplicación a crear.

Un evento puede definirse como "un cambio significativo de estado". Por ejemplo, una petición de vacaciones puede estar en estado de "en espera" o de "aprobado". Un sistema que implemente esta lógica podría tratar este cambio de estado como un evento que se pueda producir, detectar y consumir por varios componentes dentro de la arquitectura.

El patrón de arquitectura EDA puede aplicarse en el diseño y la implementación de aplicaciones que transmitan eventos a lo largo de diferentes objetos (componentes y servicios débilmente acoplados, a ser posible). Un sistema dirigido por eventos normalmente dispondrá de emisores de eventos (denominados también como '*Agentes'*) y consumidores de eventos (denominados también como 'sumidero' o *sink*). Los *sinks* tienen la responsabilidad de aplicar una reacción tan pronto como se presente un evento. Esa reacción puede o no ser proporcionada completamente por el propio *sink*. Por ejemplo, el *sink* puede tener la responsabilidad de filtrar, transformar y mandar el evento a otro componente o él mismo puede proporcionar una reacción propia a dicho evento.

El construir aplicaciones y sistemas alrededor del concepto de una orientación a eventos permite a dichas aplicaciones reaccionar de una forma mucho más natural y cercana al mundo real, porque los sistemas orientados a eventos son, por diseño, más orientados a entornos asíncronos y no predecibles (El ejemplo típico serían los *Workflows*, pero no solamente debemos encasillar EDA en *Workflows*).

EDA (*Event-Driven Architecture*), puede complementar perfectamente a una arquitectura *N-Layer DDD* y a arquitecturas orientadas a servicios (SOA) porque la lógica del dominio y los servicios-web pueden activarse por disparadores relacionados con eventos de entrada. Este paradigma es especialmente útil cuando el *sink* no proporciona él mismo la reacción/ejecución esperada.

Esta 'inteligencia' basada en eventos facilita el diseño e implementación de procesos automatizados de negocio así como flujos de trabajo orientados al usuario (*Human Workflows*); incluso es también muy útil para procesos de maquinaria, dispositivos como sensores, actuadores, controladores, etc. que pueden detectar cambios en objetos o condiciones para crear eventos que puedan entonces ser procesados por un servicio o sistema.

Por lo tanto, **se puede llegar a implementar EDA en cualquier área orientada a eventos, bien sean *Workflows*, <u>procesos de reglas del Dominio</u>, o incluso capas de presentación basadas en eventos (como MVP y M-V-VM), etc**.

EDA también está muy relacionado con el patrón **CQRS** (*Command and Query Responsibility Segregation*) que introduciremos posteriormente.

Finalmente, resaltar que en la presente propuesta de Arquitectura, así como en nuestra aplicación ejemplo publicada en *CODEPLEX*, no estamos haciendo uso de **EDA** (*Event-Driven Architecture*), simplemente lo introducimos aquí como un aspecto de arquitectura para escenarios avanzados hacia los que se puede evolucionar. Es también posible que en siguientes versiones lleguemos a evolucionar esta arquitectura hacia **EDA**.

4.- ACCESO DUAL A FUENTES DE DATOS

En la mayoría de los sistemas, lo usuarios necesitan ver datos y realizar todo tipo de búsquedas, ordenaciones y filtros, al margen de las operaciones transaccionales y/o de actualización de datos.

Para realizar dichas consultas cuyo objetivo es únicamente visualizar (informes, consultas, etc.), podríamos hacer uso de las mismas clases de lógica del dominio y repositorios de acceso a datos relacionados que utilizamos para operaciones transaccionales (en muchas aplicaciones lo haremos así), sin embargo, si se busca la máxima optimización y rendimiento, probablemente esta no sea la mejor opción.

En definitiva, mostrar información al usuario no está ligado a la mayoría de los comportamientos del dominio (reglas de negocio), ni a problemáticas de tipos de concurrencia en las actualizaciones (Gestión de Concurrencia Optimista ni su gestión de excepciones), ni por lo tanto tampoco a entidades desconectadas *self-tracking*, necesarias para la gestión de concurrencia optimista, etc. Todas estas problemáticas impactan en definitiva en el rendimiento puro de las consultas y lo único que queremos realmente hacer en este caso es realizar consultas con muy buen rendimiento. Incluso aunque tengamos requerimientos de seguridad u otro tipo que sí estén también relacionados con las consultas puras de datos (informes, listados, etc.), esto también se puede implementar en otro sitio.

Por supuesto que se puede hacer uso de un único modelo y acceso a las fuentes de datos, pero a la hora de escalar y optimizar al máximo el rendimiento, esto no se podrá conseguir. En definitiva *"un cuchillo está hecho para la carne y una cuchara para la sopa"*. Con mucho esfuerzo podremos cortar carne con una cuchara, pero no es lo más óptimo, ni mucho menos.

Es bastante normal que los Arquitectos de Software y los Desarrolladores definan ciertos requerimientos, a veces innecesarios, de una forma inflexible, y que además, a nivel de negocio no se necesitan. Este es probablemente uno de esos casos. La decisión de utilizar las entidades del modelo del dominio para solo mostrar información (solo visualización, informes, listados, etc.) es realmente algo auto-impuesto por los desarrolladores o arquitectos, pero no tiene por qué ser así.

Otro ejemplo diferente, es el hecho de que en muchos sistemas multiusuario, los cambios no tienen por qué ser visibles inmediatamente al resto de los usuarios. Si esto es así, ¿por qué hacer uso del mismo dominio, repositorios y fuentes de datos transaccionales?. Si no se necesita del comportamiento de esos dominios, ¿por qué pasar a través de ellos?. Es muy posible, por ejemplo, que las consultas (para informes, y consultas solo visualización) sean mucho más optimas en muchos casos si se utiliza una segunda base de datos basada en cubos, BI (p.e. SQL Server OLAP, etc.) y que para acceder a ello se utilice el mecanismo más sencillo y ligero para realizar consultas (una simple librería de acceso a datos, probablemente para conseguir el máximo rendimiento, el mejor camino no sea un ORM.).

En definitiva, en algunos sistemas, la mejor arquitectura podría estar basada en dos pilares internos de acceso a datos:

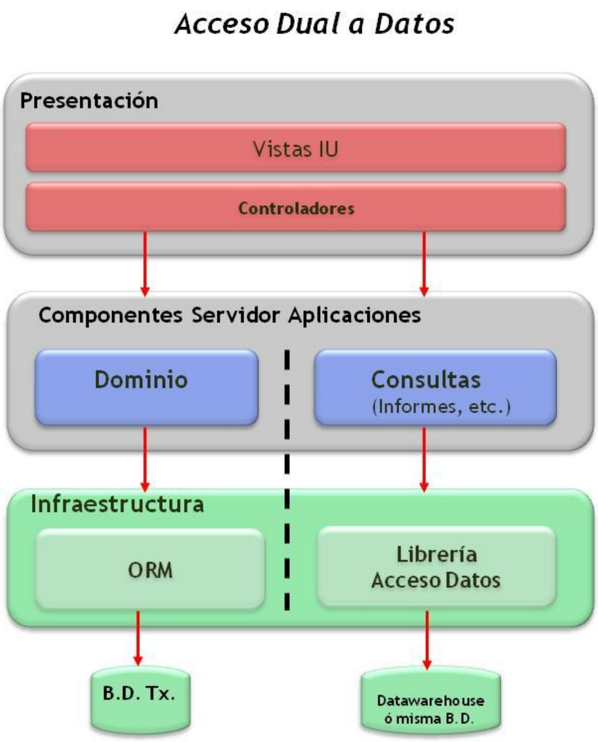


Figura 20.- Acceso Dual a Datos

Lo importante a resaltar de este modelo/arquitectura es que el pilar de la derecha se utiliza solo para consultas puras (informes, listados, visualizaciones). En cambio, el pilar de la izquierda (Dominio+ORM) seguirá realizando consultas para casos en los que dichos datos consultados pueden ser modificados por el usuario, utilizando *databinding*, etc.

Así mismo, la viabilidad de disponer o no de una base de datos diferente (incluso de tipo diferente, relacional vs. cubos), depende mucho de la naturaleza de la aplicación, pero en caso de ser viable, es la mejor opción, pues las escrituras no interferirán nunca con las 'solo lecturas', esto finalmente maximiza al máximo la escalabilidad y el rendimiento de cada tipo de operación. Sin embargo, en este caso se requerirá de algún tipo de sincronización de datos entre las diferentes bases de datos.

En definitiva, el objetivo final es *"situar todo el código en cada parte adecuada del sistema, de una forma granularizada, focalizada y que se pueda probar de forma automatizada"*.

5.- NÍVELES FÍSICOS EN DESPLIEGUE (*TIERS*)

Los Niveles representan separaciones físicas de las funcionalidades de presentación, negocio y datos de nuestro diseño en diferentes máquinas, como servidores (para lógica de negocio y bases de datos) y otros sistemas (PCs para capas de presentación remotas, etc.). Patrones de diseño comunes basados en niveles son "*2-Tier*", "*3-Tier*" y "*N-Tier*".

2-Tier

Este patrón representa una estructura básica con dos niveles principales, un nivel cliente y un servidor de bases de datos. En un escenario web típico, la capa de presentación cliente y la lógica de negocio co-existen normalmente en el mismo servidor, el cual accede a su vez al servidor de bases de datos. Así pues, en escenarios Web, el nivel cliente suele contener tanto la capa de presentación como la capa de lógica de negocio, siendo importante por mantenibilidad que se mantengan dichas capas lógicas internamente.

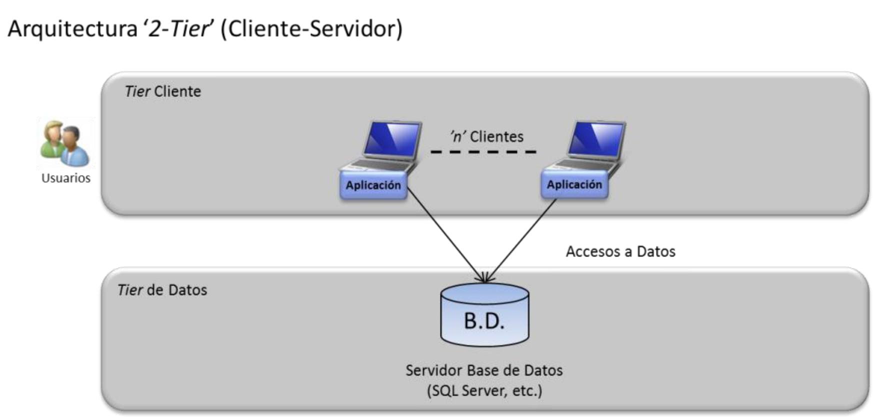


Figura 21.- Nivel/Tier Cliente

3-Tier

En un diseño de patrón "3-Tier", el usuario interacciona con una aplicación cliente desplegada físicamente en su máquina (PC, normalmente). Dicha aplicación cliente se comunica con un servidor de aplicaciones (Tier Web/App) que tendrá embebidas las capas lógicas de lógica de negocio y acceso a datos. Finalmente, dicho servidor de

aplicaciones accede a un tercer nivel (Tier de datos) que es el servidor de bases de datos. Este patrón es muy común en todas las aplicaciones Rich-Client, RIA y OBA. También en escenarios Web, donde el cliente sería 'pasivo', es decir, un simple navegador.

El siguiente gráfico ilustra este patrón "**3-Tier**" de despliegue:

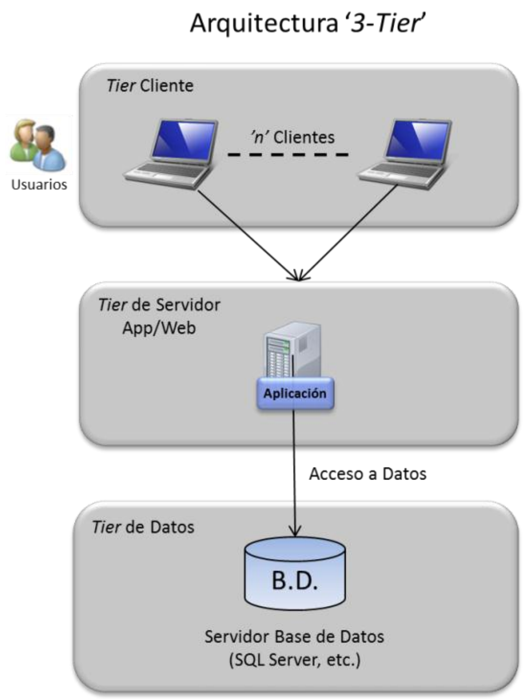


Figura 22.- Nivel/Tier Cliente

N-Tier

En este escenario, el servidor Web (que contiene la capa de lógica de presentación) se separa físicamente del servidor de aplicaciones que implementa ya exclusivamente lógica de negocio y acceso a datos. Esta separación se suele hacer normalmente por razones de políticas de seguridad de redes, donde el servidor Web se despliega en una red perimetral y accede al servidor de aplicaciones que está localizado en una subred diferente, separados probablemente por un firewall. También es común que exista un segundo *firewall* entre el nivel cliente y el nivel Web.

La siguiente figura ilustra el patrón de despliegue **"N-Tier":**

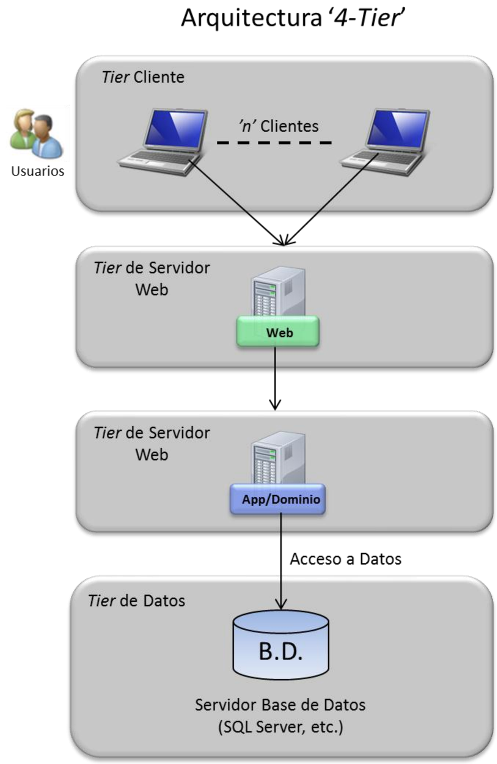


Figura 23.- Nivel/Tier Cliente

Elección de niveles/*tiers* en la arquitectura

La elección de niveles/*tiers* separando capas lógicas de nuestra aplicación en niveles físicos separados, impacta en el rendimiento de las aplicaciones (debido a la latencia de las comunicaciones remotas entre los diferentes niveles), si bien, puede beneficiar a la escalabilidad al distribuir la carga entre diferentes servidores. También puede mejorar la seguridad el separar los componentes más sensibles de la aplicación a diferentes redes. Sin embargo, hay que tener siempre presente que la adición de niveles/*tiers* incrementa la complejidad de los despliegues y en ocasiones impacta sobre el rendimiento, por lo que no se deben añadir más niveles de los necesarios.

En la mayoría de los casos, se debe localizar todo el código de la aplicación en un mismo servidor o mismo nivel de servidores balanceados. Siempre que se haga uso de comunicaciones remotas, el rendimiento se ve afectado por la latencia de las comunicaciones así como por el hecho de que los datos deben serializarse para viajar por la red. Sin embargo, en algunos casos podemos necesitar dividir funcionalidad en diferentes niveles de servidores a causa de restricciones de seguridad o requerimientos de escalabilidad. En esos casos, siempre es deseable elegir protocolos de comunicación optimizados para maximizar el rendimiento (TCP vs. HTTP, etc.).

Considera el patrón “2-Tier” si:

- **Aplicación Web**. Se quiere desarrollar una aplicación Web típica, con el máximo rendimiento y sin restricciones de seguridad de redes. Si se requiere aumentar la escalabilidad, se clonaría el Servidor Web en múltiples servidores balanceados.

- **Aplicación Cliente-Servidor**. Se quiere desarrollar una aplicación cliente-servidor que acceda directamente a un servidor de bases de datos. Este escenario es muy diferente, pues todas las capas lógicas estarían situadas en un nivel cliente que en este caso sería el PC cliente. Esta arquitectura es útil cuando se requiere un rendimiento muy alto y accesos rápidos a la base de datos, sin embargo, las arquitecturas cliente-servidor ofrecen muchos problemas de escalabilidad y sobre todo de mantenimiento y detección de problemas, pues se mueve toda la lógica de negocio y acceso a datos al nivel del PC cliente del usuario, estando a merced de las diferentes configuraciones de cada usuario final. Este caso no se recomienda en la mayoría de las ocasiones.

Considera el patrón “3-Tier” si:

- Se quiere desarrollar una aplicación “3-Tier” con cliente remoto ejecutándose en la máquina cliente de los usuarios (“Rich-Client”, RIA, OBA, etc.) y un servidor de aplicaciones con servicios web publicando la lógica de negocio.

- Todos los servidores de aplicación pueden estar localizados en la misma red.

- Se está desarrollando una aplicación de tipo ‘intranet’ donde los requerimientos de seguridad no exigen separar la capa de presentación de las capas de negocio y acceso a datos.

- Se quiere desarrollar una aplicación Web típica, con el máximo rendimiento.

Considera el patrón “N-Tier” si:

- Existen requerimientos de seguridad que exigen que la lógica de negocio no pueda desplegarse en la red perimetral donde están situados los servidores de capa de presentación.

- Se tienen código de aplicación muy pesado (hace uso intensivo de los recursos del servidor) y para mejorar la escalabilidad se separa dicha funcionalidad de componentes de negocio a otro nivel de servidores.

CAPÍTULO

4

Capa de Infraestructura de Persistencia de Datos

1.- CAPA DE INFRAESTRUCTURA DE PERSISTENCIA DE DATOS

Esta sección describe la arquitectura de la Capa de Persistencia de datos. Esta Capa de Persistencia, siguiendo las tendencias de Arquitectura DDD, forma realmente parte de las Capas de Infraestructura tal y como se define en la Arquitectura DDD propuesta por Eric-Evans, puesto que estará finalmente ligada a ciertas tecnologías específicas (de acceso a datos, en este caso). Sin embargo, debido a la importancia que tiene el acceso a datos en una aplicación y al cierto paralelismo y relación con la Capa de Dominio, en la presente guía de Arquitectura se propone que tenga preponderancia e identidad propia con respecto al resto de aspectos de infraestructura (ligados también a tecnologías concretas) a los cuales llamamos 'Infraestructura Transversal' y que se explican en otro capítulo dedicado a ello. Además, de esta forma estamos también alineados con Arquitecturas N-Capas tradicionales donde se trata a la "*Capa de Acceso a Datos*" como un ente con identidad propia (aunque no sean exactamente los mismos conceptos de capa).

Así pues, este capítulo describe las guías clave para diseñar una Capa de Persistencia de datos de una aplicación. Expondremos cómo esta capa encaja y se sitúa en la Arquitectura marco propuesta 'N-Capas con Orientación al Dominio', los patrones y componentes que normalmente contiene, y los problemas a tener en cuenta cuando se diseña esta capa. Se cubre por lo tanto guías de diseño, pasos recomendados a tomar, y patrones de diseño importantes. Finalmente, y como sub-capítulo anexo y 'separable', se explican las opciones tecnológicas y la implementación concreta con tecnología Microsoft.

Los componentes de persistencia de datos proporcionan acceso a datos que están hospedados dentro de las fronteras de nuestro sistema (p.e. nuestra base de datos principal), pero también datos expuestos fuera de las fronteras de nuestro sistema,

como Servicios Web de sistemas externos. Contiene, por lo tanto, componentes de tipo 'Repositorio' que proporcionan la funcionalidad de acceder a datos hospedados dentro de las fronteras de nuestro sistema o bien 'Agentes de Servicio' que consumirán Servicios Web que expongan otros sistemas *back-end* externos. Adicionalmente, esta capa dispondrá normalmente de componentes/clases base con código reutilizable por todas las clases 'repositorio'.

2.- ARQUITECTURA Y DISEÑO LÓGICO DE LA CAPA DE PERSISTENCIA DE DATOS I

En el siguiente diagrama se muestra cómo encaja típicamente esta capa de Persistencia de Datos dentro de nuestra arquitectura *N-Layer Domain Oriented*:

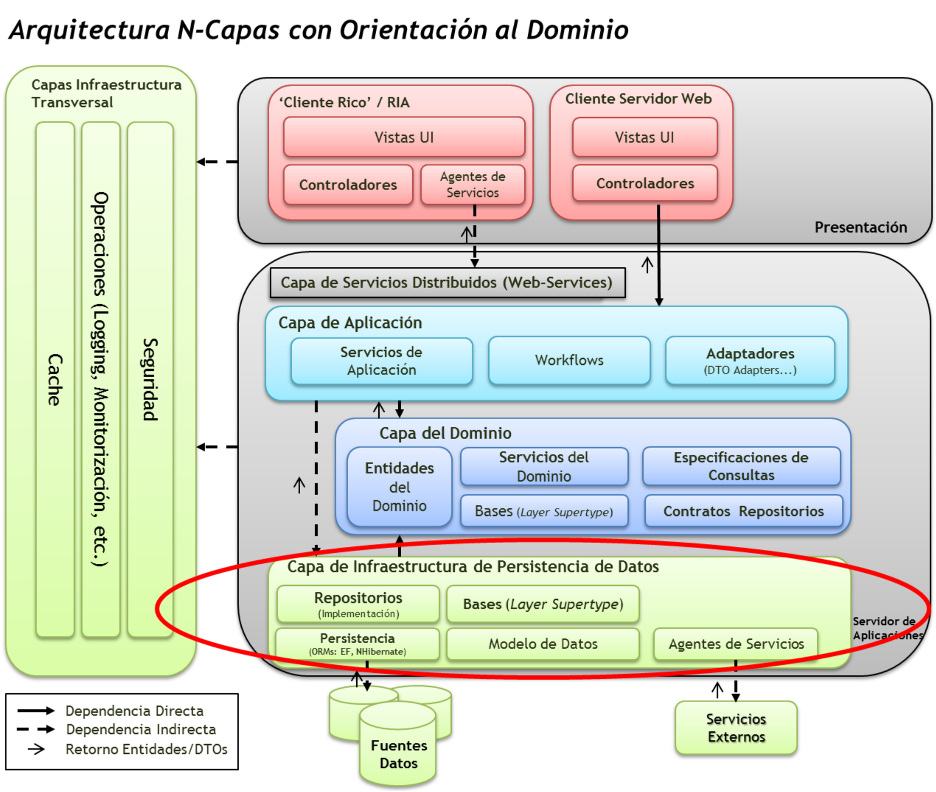


Figura 1.- Arquitectura N-Capas con Orientación al Dominio

2.1.- Elementos de la Capa de Persistencia y Acceso a Datos

La Capa de Persistencia de datos suele incluir diferentes tipos de componentes. A continuación explicamos brevemente las responsabilidades de cada tipo de elemento propuesto para esta capa:

2.1.1.-Repositorios (Repository pattern)

Estos componentes son muy similares en algunos aspectos a los componentes de 'Acceso a Datos' (DAL) de Arquitecturas tradicionales N-*Layered*. Básicamente, los Repositorios son clases/componentes que encapsulan la lógica requerida para acceder a las fuentes de datos de la aplicación. Centralizan por lo tanto funcionalidad común de acceso a datos de forma que la aplicación pueda disponer de un mejor mantenimiento y desacoplamiento entre la tecnología y la lógica de las capas 'Aplicación' y 'Dominio'. Si se hace uso de tecnologías base tipo O/RM (*Object/Relational Mapping frameworks*), se simplifica mucho el código a implementar y el desarrollo se puede focalizar exclusivamente en los accesos a datos y no tanto en la tecnología de acceso a datos (conexiones a bases de datos, sentencias SQL, etc.) que se hace mucho más transparente en un O/RM. Por el contrario, si se utilizan componentes de acceso a datos de más bajo nivel, normalmente es necesario disponer de clases utilidad propias que sean reutilizables para tareas similares de acceso a datos.

Pero es fundamental diferenciar entre un objeto '*Data Access*' (utilizados en muchas arquitecturas tradicionales N-Layer) de un Repositorio. La principal diferencia radica en que un objeto '*Data Access*' realiza directamente las operaciones de persistencia y acceso a datos directamente contra el almacén (normalmente una base de datos). Sin embargo, un Repositorio "registra" en memoria (un contexto del almacén) los datos con los que está trabajando e incluso las operaciones que se quieren hacer contra el almacén (normalmente base de datos), pero estas no se realizarán hasta que desde la capa de Aplicación se quieran efectuar esas 'n' operaciones de persistencia/acceso en una misma acción, todas a la vez. Esta decisión de 'Aplicar Cambios' que están en memoria sobre el almacén real con persistencia, está basado normalmente en el patrón 'Unidad de Trabajo' o '*Unit of Work*', que se explicará en detalle en el capítulo de 'Capa de Aplicación'. Este patrón o forma de aplicar/efectuar operaciones contra los almacenes, en muchos casos puede aumentar el rendimiento de las aplicaciones y en cualquier caso, reduce las posibilidades de que se produzcan inconsistencias. También reduce los tiempos de bloqueos en tabla de transacciones porque las operaciones de una transacción se van a ejecutar mucho más inmediatamente que con otro tipo de acceso a datos que no agrupe las acciones contra el almacén.

Patrón *Repository*

'*Repository*' es una de las formas bien documentadas de trabajar con una fuente de datos. Otra vez *Martin Fowler* en su libro PoEAA describe un repositorio de la siguiente forma:

> "*Un repositorio realiza las tareas de intermediario entre las capas de modelo de dominio y mapeo de datos, actuando de forma similar a una colección en memoria de objetos del dominio. Los objetos cliente construyen de forma declarativa consultas y las envían a los repositorios para que las satisfagan. Conceptualmente, un repositorio encapsula a un conjunto de objetos almacenados en la base de datos y las operaciones que sobre ellos pueden realizarse, proveyendo de una forma más cercana a la orientación a objetos de la vista de la capa de persistencia. Los repositorios, también soportan el objetivo de separar claramente y en una dirección la dependencia entre el dominio de trabajo y el mapeo o asignación de los datos*".

Este patrón, es uno de los más habituales hoy en día, sobre todo si pensamos en *Domain Driven Design*, puesto que nos permite de una forma sencilla, hacer que nuestras capas de datos sean '*testables'* y trabajar de una forma más simétrica a la orientación a objetos con nuestros modelos relaciones . *Microsoft Patterns & Practices* dispone de una implementación de este patrón, *Repository Factory*, disponible para descarga en *CodePlex* (Recomendamos solo su estudio, no su uso, puesto que hace uso de versiones de tecnologías y *frameworks* algo anticuados).

Así pues, para cada tipo de objeto que necesite acceso global (normalmente por cada Entidad raíz de un *Agregado*), se debe crear un objeto (Repositorio) que proporcione la apariencia de una colección en memoria de todos los objetos de ese tipo. Se debe establecer acceso mediante un interfaz bien conocido, proporcionar métodos para consultar, añadir, modificar y eliminar objetos, que realmente encapsularán la inserción o eliminación de datos en el almacén de datos. Proporcionar métodos que seleccionen objetos basándose en ciertos criterios de selección y devuelvan objetos o colecciones de objetos instanciados (entidades del dominio) con los valores de dicho criterio, de forma que encapsule el almacén real (base de datos, etc.) y la tecnología base de consulta.

Es importante volver a recalcar que se deben definir REPOSITORIOS solo para las entidades lógicas principales (En DDD serán los *AGGREGATE roots o bien ENTIDADES sencillas*), no para cualquier tabla de la fuente de datos.

Todo esto hace que en capas superiores se mantenga focalizado el desarrollo en el modelo y se delega todo el acceso y persistencia de objetos a los REPOSITORIOS.

Tabla 1.- Guía de Arquitectura Marco

Regla Nº: D4.	**Diseñar e implementar Sub-Capa de Repositorios para persistencia de datos y acceso a datos**

Normas

- Para encapsular la lógica de persistencia de datos, se diseñarán e implementarán clases de tipo 'Repositorio'. Los Repositorios estarán normalmente apoyados sobre *frameworks* de mapeo de datos, tipo ORM.

Ventajas del uso de Repositorios

→ Se presenta al desarrollador de las Capas de Aplicación y Dominio un modelo más sencillo para obtener 'objetos/entidades persistidos' y gestionar su ciclo de vida.

→ Desacopla a la capa de APLICACIÓN y DOMINIO de la tecnología de persistencia, estrategias de múltiples bases de datos, o incluso de múltiples fuentes de datos.

→ Permiten ser sustituidos fácilmente por implementaciones falsas de acceso a datos (fake), a ser utilizadas en *testing* (pruebas unitarias sobre todo) de la lógica del dominio. Normalmente se suelen sustituir por colecciones en memoria, generadas '*hard-coded*'.

Referencias

- *Patrón 'Repository'. Por Martin Fowler.*
http://martinfowler.com/eaaCatalog/repository.html
- *Patrón 'Repository'. Por Eric Evans en su libro DDD.*

Como se puede observar en el gráfico siguiente, tendremos una clase de *Repository* por cada entidad lógica de datos (entidad principal o también llamadas en DDD como *AGGREGATE roots*, que puede estar representada/persistida en la base de datos por una o más tablas, pero solo uno de los tipos de 'objeto' será el tipo raíz por el que se canalizará:

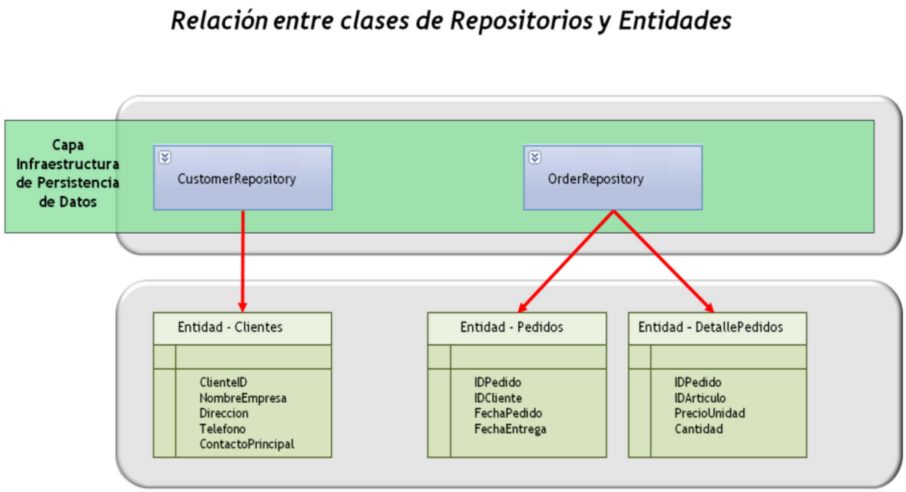


Figura 2.- Relación entre clases de Repositorios y Entidades

Relativo al concepto '*Aggregate Root*', en el ejemplo anterior, el objeto raíz sería '*Order*'. O incluso, en el caso de Clientes y su repositorio '*CustomerRepository*', si la dirección no es una entidad en la aplicación (porque no requiera identidad propia), el objeto raíz para obtener direcciones debería ser siempre el objeto repositorio de Clientes, '*CustomerRepository*'.

Tabla 2.- Guía de Arquitectura Marco

Regla Nº: D5.

Clases Repository (clases de persistencia y acceso a datos) como únicos responsables de interlocución con almacenes

○ **Norma**

- **Dentro de la Arquitectura DDD definida de un proyecto, los únicos interlocutores con los Almacenes (típicamente tablas de bases de datos, pero pueden ser también otro tipo de almacenes), serán los Repositorios.** Esto no impide que en sistemas externos a la arquitectura *Domain Oriented*, si se podría acceder por otro camino a dichas tablas de B.D., por ejemplo para integrar la B.D. transaccional con un BI (*Business Intelligence*) o generar informes con otras herramientas, entonces si es admitido, lógicamente, que se acceda por otro camino que no tenga nada que ver con nuestros Repositorios.

Tabla 3.- Guía de Arquitectura Marco

Regla Nº: D6.	**Implementar patrón "Super-Tipo de Capa" (Layer Supertype) para la Sub-Capa de Repositorios**

○ **Recomendaciones**

- Es usual y muy útil disponer de 'clases base' de cada capa para agrupar y reutilizar métodos comunes que no queremos tener duplicados en diferentes partes del sistema. Este sencillo patrón se le llama "*Layer SuperType*".

- Es especialmente útil para reutilizar código de acceso a datos que es similar para las diferentes entidades de datos.

Referencias

Patrón 'Layer Supertype'. Por Martin Fowler.
http://martinfowler.com/eaaCatalog/layerSupertype.html

Relación de 'Especificaciones de Consultas' con Repositorios

Las Especificaciones de consultas son una forma abierta y extensible de definir criterios de consulta. Son definidas desde la Capa de Dominio, pero sin embargo, son aplicadas en los Repositorios de la capa de Infraestructura de acceso a datos. Debido a que la definición se realiza en la Capa de *Dominio* y se utilizan en la capa de *Aplicación*, se explican en más detalle es el capítulo dedicado a la Capa de Dominio.

2.1.2.-Modelo de Datos

Este concepto existe a veces en la implementación de la Capa de Persistencia para poder definir e, incluso visualizar gráficamente el modelo de datos 'entidad-relación' de la aplicación. Este concepto suele ser proporcionado completamente por la tecnología O/RM concreta que se utilice, por lo que está completamente ligado a una infraestructura/tecnología específica (p.e. *Entity Framework* proporciona una forma de

definir un modelo entidad-relación o incluso de crearlo a partir de una base de datos existente).

2.1.3.- Tecnología de Persistencia (O/RM, etc.)

Es simplemente la capa de infraestructura/tecnología utilizada internamente por nuestros Repositorios. Normalmente será, por lo tanto, la propia tecnología que hayamos escogido, bien un O/RM como *Entity Framework* o *NHibernate*, o simplemente tecnología de más bajo nivel como ADO.NET. Pero en este caso estaríamos hablando ya de tecnología, por lo que esto se explica en detalle en el sub-capítulo de 'Implementación de Capa de Persistencia', en la última parte del presente capítulo.

2.1.4.- Agentes de Servicios Distribuidos externos

Cuando un componente del dominio debe acceder a datos proporcionados por un servicio distribuido externo (p.e. un Servicio-Web), debemos implementar código que gestione la semántica de comunicación con dicho servicio en particular. Estos Agentes de Servicio implementan precisamente componentes de acceso a datos que encapsulan y aíslan los requerimientos de los Servicios distribuidos e incluso pueden soportar aspectos adicionales como cache, soporte off-line y mapeos básicos entre el formato de datos expuesto en los Servicios distribuidos externos y el formato de datos requerido/utilizado por nuestra aplicación.

2.2.- Otros patrones de acceso a datos

Los patrones que explicamos a continuación ayudan a comprender las diferentes posibilidades de estrategias de accesos a datos y por lo tanto, ayudan a comprender mejor las opciones elegidas por la presente guía de arquitectura y diseño.

Aunque pueda parecer extraño, después de tantos años de avances tecnológicos, acceder a datos es un elemento importante y sumamente delicado dentro de nuestros desarrollos, tan delicado como para llevar al 'traste' un proyecto entero, bien por tiempos de productividad como por la solución al problema en sí. La gran cantidad de técnicas y patrones que existen hoy en día con respecto al acceso a datos no hacen más que agregar un grado de confusión mayor a muchos programadores. Por supuesto, cada una de las posibles técnicas agrega elementos favorables y otros no

tanto, por lo que una buena elección de la misma es un factor de éxito importante en la vida de un proyecto.

Siempre viene bien recordar ciertos patrones conocidos y bien documentados, estos sin duda, nos ayudarán a entender la filosofía de la presente guía de Arquitectura y Diseño.

2.2.1.- Active Record

Sin duda, este es uno de los patrones más conocidos y usados. Y como suele ocurrir a menudo con los patrones, a veces no conocemos el nombre dado pero si estamos hartos de haberlo usado. Si recurrimos a *Martin Fowler* en su libro "*Patterns Of Enterpise Application Architecture:PoEAA*", podremos entender un objeto 'Active Record' como un objeto que transporta no solamente datos sino también el comportamiento. Es decir, un Active Record deposita la lógica de su persistencia dentro del propio dominio del objeto.

Este patrón de diseño está puesto en práctica en muchas implementaciones de lenguajes dinámicos como *Ruby* y es usado ampliamente hoy en día por la comunidad de desarrolladores. Dentro de la tecnología .NET, hoy en día existen numerosas implementaciones como *Castle Active Record*, *.NetTiers Application Framework* o *LLBLGenPro* por poner algunos ejemplos.

Sin embargo, uno de los inconvenientes más grandes de este patrón viene de su propia definición, al no separar conceptualmente el transporte de datos de sus mecanismos de persistencia. Si pensamos en arquitecturas orientadas a servicios dónde precisamente una separación entre los contratos de datos y las operaciones sobre los mismos es uno de los pilares de SOA, veremos que una solución como esta (*Active Record*) no es apropiada y en muchas ocasiones es extremadamente difícil de implementar y mantener. Otro ejemplo donde una solución basada en '*Active Record*' no es en principio una buena elección es aquella en la que no hay una relación 1:1 entre las tablas de la base de datos y los objetos Active Record dentro de nuestros modelos de dominio, o bien la lógica que estos objetos tienen que disponer es algo compleja.

2.2.2.- Table Data Gateway

Este patrón, también perfectamente documentado en **PoEAA,** puede verse como un refinamiento del anterior, intentando separar el propio transporte de datos de las operaciones sobre el mantenimiento de los mismos. Para muchos programadores esto supone una mejora, al delegar en un intermediario o *gateway* todo el trabajo de interacción con la base de datos. Al igual que Active Record, este patrón funciona bien

cuando las relaciones de nuestras entidades están asociadas en forma de 1:1 con respecto a las tablas de la base de datos, sin embargo, cuando dentro de nuestro dominio de entidades deseamos realizar elementos más complejos como herencias, tipos complejos o asociados etc., este modelo pierde su fuerza y en muchas ocasiones su sentido.

2.2.3.- Data Mapper

Si pensamos en los dos patrones anteriores, veremos como ambos padecen el acoplamiento de las entidades del dominio con respecto al modelo de datos. La realidad es que los modelos de objetos y los modelos de datos disponen de diferentes mecanismos para estructurar datos, y en muchas ocasiones, hacen que los desarrolladores no puedan aprovechar todos los conocimientos de orientación a objetos cuando se trabaja con bases de datos o bien se vea penalizado nuestro desarrollo por un determinado modelo relacional.

Las diferencias entre los modelos relacionales y los modelos de dominio son muchas y a esta situación suele denominársela como desajuste de impedancias o '*impedance mistmach*'. Un ejemplo bueno es el tratamiento de las relaciones en ambos mundos. En los modelos relacionales, las relaciones se establecen mediante la duplicación de los datos en distintas tablas, de tal forma que, si deseamos relacionar una tupla de una tabla B con una tupla de una tabla A, deberemos establecer en la tabla B una columna que contenga un valor que permita identificar al elemento de la tabla A con el que la queremos relacionar. Sin embargo, en los lenguajes de programación orientados a objetos como C# o Visual Basic las relaciones no necesitan apoyarse en la duplicidad de datos; por seguir con el ejemplo bastaría con poner una referencia en el objeto B al objeto A con el que se desea establecer una relación, que en el mundo orientado a objetos recibe el nombre de asociación.

El patrón *Data Mapper* tiene como objetivo separar las estructuras de los objetos de las estructuras de los modelos relacionales y realizar la transferencia de datos entre ambos. Con el uso de un *Data Mapper*, los objetos que consumen los componentes 'DataMapper', son ignorantes del esquema presente en la base de datos y, por supuesto, no necesitan hacer uso de código SQL.

2.2.4.- Lista de patrones para las capas de Persistencia de Datos

En la siguiente tabla se enumeran los patrones posibles para la capa de persistencia de datos.

Tabla 4.- Categorías/Patrones

Patrones
• Active Record
• Data Mapper
• Query Object
• Repository
• Row Data Gateway
• Table Data Gateway
• Table Module

Referencias adicionales

Información sobre Domain Model, Table Module, Coarse-Grained Lock, Implicit Lock,Transaction Script, Active Record, Data Mapper, Optimistic Offline Locking, Pessimistic Offline Locking, Query Object, Repository, Row Data Gateway, and Table Data Gateway patterns, ver:
"Patterns of Enterprise Application Architecture (P of EAA)" en
http://martinfowler.com/eaaCatalog/

3.- PRUEBAS EN LA CAPA DE INFRAESTRUCTURA DE PERSISTENCIA DE DATOS

Al igual que cualquiera de los demás elementos de una solución, nuestra Capa de Persistencia de Datos es otra superficie que también debería estar cubierta por un conjunto de pruebas y, por supuesto, cumplir los mismos requisitos que se le exigen en el resto de capas o de partes de un proyecto. La implicación de una dependencia externa como una base de datos tiene unas consideraciones especiales que deben de ser tratadas con cuidado para no incurrir en algunos anti-patrones comunes en el diseño de pruebas unitarias, en concreto, deberían evitarse los siguientes defectos en las pruebas creadas.

Anti-patrones a evitar:

- **Pruebas erráticas (*Erratic Test*).** Una o más pruebas se comportan de forma incorrecta, algunas veces las pruebas se ejecutan de forma correcta y otras veces no. El principal impacto de este tipo de comportamientos se debe al tratamiento que sobre los mismos se tiene, puesto que suelen ser ignoradas e internamente podrían esconder algún defecto de código que no se trata.

- **Pruebas lentas (*Slow Tests*).** Las pruebas necesitan una gran cantidad de tiempo para llevarse a cabo. Este síntoma, por lo general, acaba provocando que los desarrolladores no ejecuten las pruebas del sistema cada vez que se realiza uno o varios cambios, lo que reduce la calidad del código al estar exento de pruebas continuas sobre el mismo y merma la productividad de las personas encargadas de mantener y ejecutar estas pruebas.

- **Pruebas oscuras (*Obscure Test*).** En muchas ocasiones, debido a ciertos elementos de inicialización de las pruebas y a los procesos de limpieza o restablecimiento de datos iniciales el sentido real de la prueba queda obscurecido y no se puede entender de un simple vistazo.

- **Pruebas irrepetibles (*Unrepeatable Test*):** El comportamiento de una prueba varía si se ejecuta inmediatamente a su finalización.

Algunas soluciones habituales para realizar pruebas en las que interviene una base de datos se pueden ver en los siguientes puntos, aunque por supuesto no son todas las existentes:

- **Asilamiento de bases de datos:** Se proporciona o se usa una base de datos diferente y separada del resto para cada uno de los desarrolladores o probadores que estén pasando pruebas que involucren a la capa de infraestructura.

- **Deshacer los cambios en la finalización de cada prueba:** En el proceso de finalización de cada prueba deshacer los cambios realizados. Para el caso de base de datos mediante el uso de transacciones. Esta alternativa tiene impacto en la velocidad de ejecución de las pruebas.

- **Rehacer el conjunto de datos en la finalización de cada prueba:** Esta alternativa consiste en rehacer el conjunto de datos al estado inicial de la prueba con el fin de que la misma se pueda repetir inmediatamente.

Tabla 5.- Pruebas en la capa de persistencia de datos

Regla Nº: D7.	**Pruebas en la capa de infraestructura de persistencia de datos**

○ **<u>Recomendaciones</u>**

- Hacer que la capa de infraestructura de persistencia pueda inyectar dependencias con respecto a quien realiza operaciones en la base de datos, de tal forma que se puede realizar una simulación, *Fake Object*, y por lo tanto poder ejecutar el conjunto de pruebas de una forma rápida y fiable.

- Si la capa de infraestructura de persistencia introduce un Layer SuperType para métodos comunes usar herencia de pruebas, si el framework usado lo permite, con el fin de mejorar la productividad en la creación de las mismas.

- Implementar un mecanismo que permita al desarrollador o probador cambiar de una forma simple si el conjunto de pruebas se ejecuta con objetos simulados o bien contra una base de datos real.

- Cuando las pruebas se ejecutan con una base de datos real debemos asegurarnos que no sufrimos los antipatrones **Unrepeatable Test** o **Erratic Test.**

Referencias

MSDN Unit Testing
http://msdn.microsoft.com/en-us/magazine/cc163665.aspx
Unit Testing Patterns
http://xunitpatterns.com/

4.- CONSIDERACIONES GENERALES DE DISEÑO DEL ACCESO A DATOS

La Capa de Persistencia y acceso a datos debe cumplir los requerimientos de la aplicación a nivel de rendimiento, seguridad, mantenibilidad y soportar cambios de requerimientos de negocio. Al diseñar la Capa de Persistencia de Datos, se debe tener en cuenta las siguientes guías de diseño:

- **Elegir una tecnología de acceso a datos apropiada.** La elección de la tecnología depende del tipo de datos que se deba gestionar y de cómo se desea manipular dentro de la aplicación. Ciertas tecnologías están mejor indicadas para ciertas tareas. Por ejemplo, aunque los OR/M están muy indicados para la mayoría de accesos a datos en una arquitectura DDD, en algunos casos es posible que no sea la mejor opción. Para dichos casos, se debe de valorar el uso de otras tecnologías.

- **Uso de abstracción para implementar desacoplamiento de la Capa de persistencia.** Esto puede realizarse mediante la definición de interfaces (contratos) de todos los Repositorios, e incluso que dichos interfaces/contratos no estén implementados dentro de la capa de persistencia (infraestructura), sino en la Capa de Dominio. En definitiva, el contrato será lo que el Dominio requiere de un Repositorio para que pueda funcionar en la aplicación y la implementación de dicho contrato es lo que estará en la Capa de Infraestructura de Persistencia de Datos. Esto además puede verse mucho más desacoplado si se hace uso de contenedores IoC para instanciar los Repositorios desde la Capa del dominio.

- **Decidir cómo gestionar las conexiones a las bases de datos**. Como norma, la Capa de Persistencia de datos será quien gestione todas las conexiones a las fuentes de datos requeridas por la aplicación. Se deben escoger métodos apropiados para guardar y proteger la información de conexión, por ejemplo, cifrando secciones de ficheros de configuración, etc.

- **Determinar cómo gestionar excepciones de datos.** Esta Capa de Persistencia de datos debe capturar y (por lo menos inicialmente) gestionar todas las excepciones relacionadas con las fuentes de datos y operaciones CRUD (*Create, Read, Update y Delete*). Las excepciones relativas a los propios datos y los errores de 'time-outs' de las fuentes de datos, deben ser gestionados en esta capa y pasados a otras capas solo si los fallos afectan a la funcionalidad y respuesta de la aplicación. Por ejemplo, excepciones de interbloqueos y problemas de conexión deben ser resueltos en la propia capa de persistencia de datos. Sin embargo, violaciones de concurrencia (Gestión Optimista de

Concurrencia) debe de propagarse hasta la capa de presentación para que el usuario resuelva el 'conflicto de negocio'.

- **Considerar riesgos de seguridad**. Esta capa debe proteger contra ataques que intenten robar o corromper datos, así como proteger los mecanismos utilizados para acceder a las fuentes de datos. Por ejemplo, hay que tener cuidado de no devolver información confidencial de errores/excepciones relativos al acceso a datos, así como acceder a las fuentes de datos con credenciales lo más bajas posibles (no con usuarios 'Administrador' de la base de datos). Adicionalmente, el acceso a los datos debe ser con consultas parametrizadas (los ORMs lo realizan así, por defecto) y nunca formando sentencias SQL por medio de concatenación de strings, para prevenir ataques de 'Inyecciones SQL'.

- **Considerar objetivos de rendimiento y escalabilidad**. Estos objetivos deben tenerse en cuenta durante el diseño de la aplicación. Por ejemplo, si se diseña una aplicación de comercio-e en Internet, el rendimiento de acceso a datos puede ser un cuello de botella. Para todos los casos donde el rendimiento y la escalabilidad es crítico, hay que considerar estrategias basadas en Cache, siempre que la lógica de negocio lo permita, por supuesto. Así mismo, se debe realizar un análisis de las consultas por medio de herramientas de *profiling* para poder determinar posibles puntos de mejora. Otras consideraciones sobre el rendimiento:

    - Hacer uso del Pool de Conexiones para lo cual es necesario minimizar el número de credenciales accediendo al servidor de base de datos.

    - En algunos casos, considerar comandos *batch* (varias operaciones en la misma ejecución de sentencia SQL).

    - Considerar uso de concurrencia optimista con datos no volátiles para mitigar el coste de bloqueos de datos en la base de datos. Esto evita la sobrecarga de bloqueos de filas en la base de datos, incluyendo la conexión que debe mantenerse abierta durante un bloqueo.

- **Mapeo de Objetos a Datos Relacionales**. En un enfoque DDD, basado en modelado de entidades como objetos del Dominio, el uso de un O/RM es normalmente la mejor elección. Los O/RMs actuales pueden reducir significativamente la cantidad de código a implementar. Para más información sobre DDD, leer el capítulo inicial de la Arquitectura. Considerar los siguientes puntos cuando se hace uso de frameworks y herramientas O/RM:

    - Las herramientas O/RM pueden permitir diseñar un modelo entidad relación y generar a partir de ello un esquema de base de datos real (A este enfoque se le llama '*Model First*') al mismo tiempo que se establece el mapeo entre objetos/entidades del dominio y la base de datos.

  - Si la base de datos ya existe, las herramientas O/RM normalmente también permiten generar el modelo de datos entidad-relación a partir de dicha base de datos existente y entonces mapear los objetos/entidades del dominio y la base de datos.

- **Procedimientos Almacenados.** En el pasado, en algunos SGBD los procedimientos almacenados proporcionaban una mejora en el rendimiento con respecto a las sentencias SQL dinámicas (Porque los procedimientos almacenados estaban compilados en cierta forma y las sentencias SQL dinámicas no lo estaban). Sin embargo, con los SGBD actuales, el rendimiento entre las sentencias SQL dinámicas y los procedimientos almacenados, es similar. Razones por las que hacer uso de procedimientos almacenados son por ejemplo el separar el acceso a datos del desarrollo, de forma que un experto en bases de datos pueda hacer *tunning* de dichos procedimientos almacenados, sin tener que conocer ni tocar el desarrollo. Sin embargo, la desventaja de hacer uso de procedimientos almacenados es que son completamente dependientes al SGBD escogido, con sentencias SQL específicas para dicho SGBD. En cambio, algunos O/RMs son capaces de generar sentencias SQL nativas para los diferentes SGBD que soportan, de forma que la portabilidad de la aplicación de un SGBD a otro sería prácticamente inmediata.

  - Algunos OR/Ms soportan el uso de procedimientos almacenados, pero al hacerlo, lógicamente se pierde la portabilidad a diferentes SGBD.
  - Si se hace uso de procedimientos almacenados, por razones de seguridad se deben utilizar parámetros tipados para impedir inyecciones SQL.
  - El *debugging* de consultas basadas en SQL dinámico y O/RM es más sencillo que realizarlo con procedimientos almacenados.
  - En general, el uso o no de procedimientos almacenados depende mucho también de las políticas de una empresa. Pero si no existen dichas políticas, la recomendación sería hacer uso de O/RMs por regla general y de procedimientos almacenados para casos especiales de consultas muy complejas y pesadas que se quieran tener muy controladas y que se puedan mejorar en el futuro por expertos en SQL.

- **Validaciones de Datos**. La gran mayoría de validaciones de datos debe realizarse en la Capa de Dominio, pues se realizarán validaciones de datos relativas a reglas de negocio. Sin embargo existen algunos tipos de validaciones de datos relativos exclusivamente a la Capa de Persistencia, como por ejemplo pueden ser:

- Validar parámetros de entrada para gestionar correctamente valores NULL y filtrar caracteres inválidos
- Validar los parámetros de entrada examinando caracteres o patrones que puedan intentar ataques de inyección SQL.
- Devolver mensajes de error informativos si la validación falla, pero ocultar información confidencial que pueda generarse en las excepciones.

- **Consideraciones de Despliegue**. En el diseño del despliegue, los objetivos de la arquitectura consisten en balancear aspectos de rendimiento, escalabilidad y seguridad de la aplicación en el entorno de producción, dependiendo de los requerimientos y prioridades de la aplicación. Considerar las siguientes guías:
  - Situar la capa de Infraestructura de Persistencia de datos en el mismo nivel físico que la Capa de Dominio para maximizar el rendimiento de la aplicación. Solo en casos de restricciones de seguridad y/o algunos casos no muy comunes de escalabilidad pueden aconsejar lo contrario. Pero prácticamente en el 100% de los casos, la Capa de Dominio y la Capa de persistencia o acceso a datos deberían estar físicamente en los mismos servidores.
  - Siempre que se pueda, localizar la capa de Infraestructura de Persistencia de datos en servidores diferentes al servidor de la Base de Datos. Si se sitúa en el mismo servidor, el SGBD estará compitiendo constantemente con la propia aplicación por conseguir los recursos del servidor (procesador y memoria), perjudicando al rendimiento de la aplicación.

4.1.- Referencias Generales

- "*.NET Data Access Architecture Guide*" - **http://msdn.microsoft.com/en-us/library/ms978510.aspx**.
- *"Concurrency Control*"- **http://msdn.microsoft.com/en-us/library/ms978457.aspx**.
- "*Data Patterns*" - **http://msdn.microsoft.com/en-us/library/ms998446.aspx**.
- "*Designing Data Tier Components and Passing Data Through Tiers*" - **http://msdn.microsoft.com/en-us/library/ms978496.aspx**.

- "*Typing, storage, reading, and writing BLOBs*" - **http://msdn.microsoft.com/en-us/library/ms978510.aspx#daag_handlingblobs**.
- "*Using stored procedures instead of SQL statements*" - **http://msdn.microsoft.com/en-us/library/ms978510.aspx**.
- "*NHibernate Forge*" community site - **http://nhforge.org/Default.aspx**.
- ADO.NET Entity Framework – En **http://msdn.microsoft.com**

5.- IMPLEMENTACIÓN EN .NET DE LA CAPA DE PERSISTENCIA DE DATOS

La explicación y definición lógica de esta capa está explicada en el capítulo anterior, por lo que en el presente capítulo no vamos a explicar los conceptos lógicos de persistencia de datos ni patrones tipo Repositorio, etc. El objetivo del presente capítulo es mostrar las diferentes opciones que tenemos a nivel de tecnología para implementar la 'Capa de Persistencia de Datos' y por supuesto, explicar la opción tecnológica elegida por defecto en nuestra Arquitectura Marco .NET 4.0, de referencia.

En el siguiente diagrama resaltamos la situación de la Capa de Persistencia de datos:

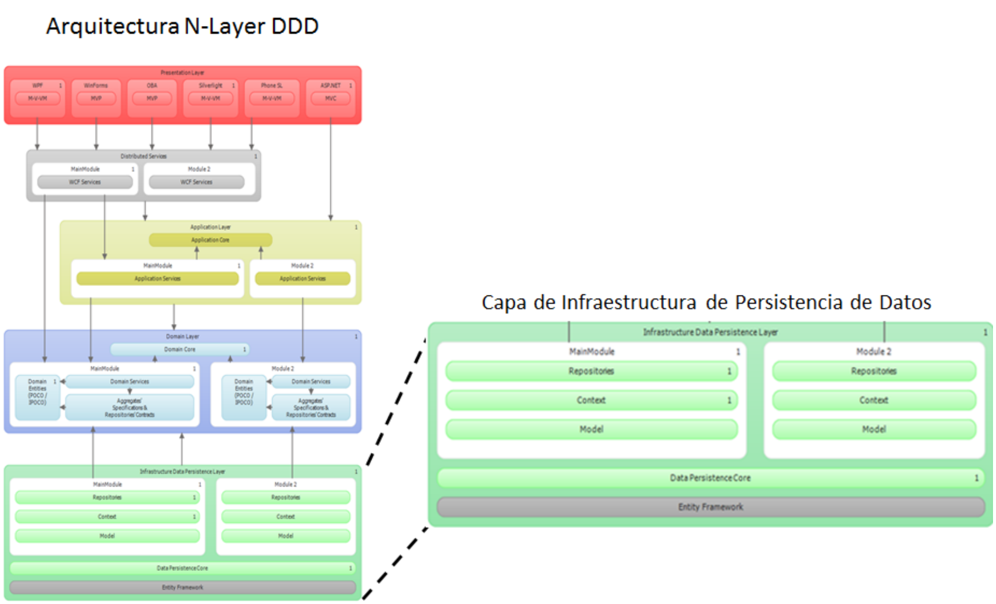


Figura 3.- Diagrama en VS.2010 de situación Capa de Persistencia

Pasos a realizar:

1.- El primer paso, será identificar las limitaciones relativas a los datos que queremos acceder, lo cual nos ayudará a seleccionar las diferentes tecnologías disponibles para implementar 'Repositorios'. ¿Estamos hablando de bases de datos relacionales? ¿Qué SGBD concretamente? ¿Se trata de otro tipo de fuente de datos?

2.- El siguiente paso es elegir la estrategia de mapeo y entonces determinar el enfoque de acceso a los datos, lo cual incluye identificar las entidades de negocio a utilizar y el formato de dichas entidades. Las entidades de negocio son realmente 'Entidades del Dominio' y quedarán definidas en la 'Capa de Dominio' y no en la presente Capa de Persistencia de Datos, sin embargo, la relación de dichas 'Entidades del Dominio' con la capa de Persistencia es muy grande y se deben tomar decisiones sobre ellas en este momento (Implementación de Capa de Persistencia), porque dependiendo de la tecnología a utilizar, se generarán/desarrollarán de una u otra forma. Así mismo, debemos determinar cómo van a conectarse los componentes 'Repositorio' a las fuentes de datos, con qué tecnología concretamente.

3.- Finalmente deberemos determinar la estrategia de gestión de errores que utilizaremos para gestionar las excepciones y errores relativos a las fuentes de datos.

5.1.- Opciones de tecnología para la Capa de Persistencia de Datos

5.2.- Selección de Tecnología de Acceso a Datos

La elección de la tecnología adecuada para acceder a los datos debe tener en cuenta el tipo de fuente de datos con la que tendremos que trabajar y cómo queremos manipular los datos dentro de la aplicación. Algunas tecnologías se ajustan mejor a ciertos escenarios. Las diferentes posibilidades y características a tener en cuenta en dicha selección, son:

- ***Entity Framework*:** (El nombre completo es *ADO.NET Entity Framework*, puesto que está basado sobre la plataforma de ADO.NET). Se debe considerar esta opción si se desea crear un modelo de datos mapeado a una base de datos relacional. A nivel superior se mapeará normalmente una clase a múltiples tablas que conformen una entidad compleja. La gran ventaja de EF es que hace transparente la base de datos con la que trabaja, pues el modelo de EF genera las sentencias SQL nativas requeridas para cada SGBD, así pues, en la mayoría de los casos sería transparente si estamos trabajando contra SQL Server,

Oracle, DB2, MySql, etc. Solo es necesario hacer uso en cada caso del proveedor de EF relativo a cada SGBD. EF es apropiado cuando se quiere hacer uso de un modelo de desarrollo ORM basado en un modelo de objetos (p.e. el de *Linq to Entities*) mapeado a un modelo relacional mediante un esquema flexible. Si se hace uso de EF, normalmente se hará uso también de:

- o ***LINQ to Entities***: Considera '*LINQ to Entities*' si se desea ejecutar consultas contra entidades fuertemente tipadas y haciendo uso del modelo orientado a objetos de la sintaxis de LINQ.

- ***ADO.NET*:** Considera hacer uso de los componentes base de ADO.NET si se necesita hacer uso de accesos a nivel más bajo de API teniendo por tanto completo control sobre ello (sentencias SQL, conexiones de datos, etc.), pero perdiendo la transparencia aportada por EF. También, en el caso de querer reutilizar inversiones existentes implementadas directamente con ADO.NET haciendo uso de lógica tradicional de acceso a datos.

- ***Building Block de Acceso a Datos de 'Microsoft P&P Enterprise Library'***: Esta librería de acceso a datos está a su vez basada también en ADO.NET, sin embargo, siempre que sea posible, es más recomendable hacer uso de *Entity Framework*, puesto que esta última forma parte de .NET Framework y en cambio este '*Building Block*' es adicional a .NET Framework y una tecnología también más antigua que EF. El propio equipo de producto de 'Microsoft P&P' recomienda EF, siempre que sea posible, antes que esta librería.

- ***ADO.NET SyncServices*:** Considera esta tecnología si se está diseñando una aplicación que debe soportar escenarios ocasionalmente desconectados/conectados o se requiere colaboración entre diferentes bases de datos.

- ***LINQ to XML***: Considera esta tecnología si en la aplicación se hace uso extensivo de documentos XML y se desea consultarlos mediante sintaxis LINQ.

5.2.1.-Otras consideraciones tecnológicas

- Si se requiere soporte a bajo nivel para las consultas y parámetros, hacer uso directamente de objetos ADO.NET.
- Si se está haciendo uso de ASP.NET como capa de presentación para mostrar información solo lectura (informes, listados, etc.) y se requiere de un rendimiento máximo, considerar el uso de ***DataReaders*** para maximizar el

rendimiento de renderizado. Los ***DataReaders*** son ideales para accesos 'solo lectura' y '*forward-only*' en los que se procesa cada fila muy rápidamente.

- Si solo se hace uso de ADO.NET y la base de datos es SQL Server, hacer uso del namesapce SqlClient para maximizar el rendimiento
- Si se hace uso de SQL Server 2008 o superior, considerar el uso de FILESTREAM para disponer de una mayor flexibilidad en el almacén y acceso a datos de tipo BLOB.
- Si se está diseñando una capa de persistencia de datos siguiendo el modelo **DDD** (Domain Driven Design), la opción más recomendada es hacer uso de un framework **O/RM** (Object/Relational Mapping) como **Entity-Framework o NHibernate**.

Tabla 6.- Guía de Arquitectura Marco

Regla Nº: I1.	***La tecnología, por defecto, para implementar la Sub-Capa de Repositorios, persistencia de datos y acceso a datos en general relacionado con la Arquitectura 'N-Layer Domain Oriented', será Microsoft ADO.NET Entity Framework***

○ **<u>Norma</u>**

- Según las consideraciones anteriores, puesto que la presente Arquitectura Marco se trata de una Arquitectura 'N-Capas Orientada al Dominio', la tecnología seleccionada para los accesos a datos relacionados con el Dominio será ENTITY FRAMEWORK, al ser el ORM sobre tecnología .NET ofrecido por Microsoft que mejor se adapta a la implementación de patrones de Diseño relacionados con DDD. Es decir, la implementación de *Repositorios* y *Unit Of Work* con EF 4.0 es directa comparado a si utilizáramos ADO.NET.
- Sin embargo, se deja la puerta abierta a utilizar otra tecnología (ADO.NET, tecnologías de *Reporting*, etc.) para aspectos paralelos que no estén relacionados con la lógica del Dominio, como puedan ser Business *Intelligence*, o simplemente consultas solo lectura para informes y/o listados que deban soportar un máximo rendimiento. Esto está precisamente explicado a nivel lógico en el capítulo inicial de la Arquitectura lógica de la presente guía.

<u>Ventajas de Entity Framework</u>

→ Independencia del SGBD. Se puede intercambiar un SGBD por otro, tipo

SQL Server, Oracle, DB2, MySql, etc.
→ Programación fuertemente tipada y con colecciones de objetos mediante 'Linq to Entities'.

Referencias
http://msdn.microsoft.com/en-us/data/aa937723.aspx

5.2.2.-Cómo obtener y persistir objetos desde el almacén de datos

Una vez identificados los requerimientos de la fuente de datos, el próximo paso es elegir una estrategia de obtención de datos y conversión a objetos (entidades del dominio) y de forma análoga, cómo transformar posteriormente dichos objetos (probablemente modificados) a datos (persistencia de objetos).

Normalmente existe un desajuste de impedancias bastante típico entre el modelo de datos orientado a objetos y los almacenes de datos relacionales, lo cual hace a veces complicado dicha 'traducción'. Existen una serie de posibilidades para afrontar dicho desajuste, pero dichas posibilidades son diferentes dependiendo de los tipos de datos, estructura, técnicas transaccionales y cómo se manipulan los datos. La mejor aproximación y más común es hacer uso de *frameworks* O/RM (como Entity Framework). Ténganse en cuenta las siguientes guías para elegir cómo obtener y persistir objetos/entidades de negocio al almacén de datos:

- Considérese el uso de un O/RM que realice dicha traducción entre las entidades del dominio y la base de datos. Si adicionalmente se está creando una aplicación y un almacén 'desde cero', se puede hacer uso del O/RM incluso para generar el esquema de la base de datos a partir del modelo lógico de datos del O/RM (Esto se puede realizar por ejemplo con *Entity Framework 4.0*). Si la base de datos es una ya existente, se pueden utilizar las herramientas del OR/M para mapear entre el modelo de datos del dominio y el modelo relacional.

- Un patrón común asociado con DDD es el modelo del dominio que se basa en gran medida en modelar entidades con objetos/clases del dominio. Esto se ha explicado a nivel lógico en capítulos anteriores de la presente guía.

- Asegurarse de que se agrupan las entidades correctamente para conseguir un máximo nivel de cohesión. Esto significa que se requieren objetos adicionales dentro del modelo de dominio y que las entidades relacionadas están agrupadas en agregados raíz ('*Aggregate roots'* en nomenclatura DDD).

- Cuando se trabaja con aplicaciones Web o Servicios-Web, se deben agrupar las entidades y proporcionar opciones para cargar parcialmente entidades del dominio con solo los datos requeridos. Esto minimiza el uso de recursos al evitar cargar modelos de dominio pesados para cada usuario en memoria y permite a las aplicaciones el gestionar un número de usuarios concurrentes mucho mayor.

5.3.- Posibilidades de Entity Framework en la Capa de Persistencia

Como se ha establecido anteriormente, **la tecnología seleccionada** en la presente guía para implementar la capa de persistencia de datos y por lo tanto, los Repositorios en nuestra arquitectura marco N-Layer DDD, **es ENTITY FRAMEWORK**.

Importante:
Antes de poder implementar los REPOSITORIOS, es necesario disponer de los tipos (clases de Entidades POCO/IPOCO) y en el caso de Arquitecturas N-Layer Domain Oriented, las Entidades de negocio deben estar localizadas en la Capa del Dominio. Sin embargo, el inicio de la creación de dichas entidades parte de un modelo de datos EF definido en la capa de Infraestructura de persistencia de datos, y ese proceso se realiza al crear la Capa de Persistencia. Pero, antes de ver cómo crear estos proyectos de la capa de Persistencia de datos, conviene tener claro qué tipo de entidades del dominio vamos a utilizar con EF (Entidades ligadas a EF vs. Entidades POCO de EF vs. Entidades Self-Tracking de EF tipo IPOCO). Ese análisis se expone en el capítulo de la Capa del Dominio, por lo que remitimos al lector a dicho capítulo para que conozca los pros y contras de cada tipo de entidad posible con EF, antes de continuar en este capítulo de implementación de Capa de Infraestructura de Persistencia de Datos.

5.3.1.-¿Qué nos aporta Entity Framework 4.0?

Tal y como hemos visto, con respecto al acceso y tratamiento de los datos tenemos muchas y variadas alternativas de enfocar una solución. Por supuesto, cada una con sus ventajas y sus inconvenientes. Una de las prioridades con respecto al desarrollo de *Entity Framework* ha sido siempre dar cabida a la mayoría de las tendencias de programación actuales y a los distintos perfiles de desarrolladores. Desde los desarrolladores a los que les gusta y se sienten cómodos y productivos con el uso de asistentes dentro del entorno hasta aquellos a los que les gusta tener un control exhaustivo sobre el código y la forma de trabajar.

Uno de los pasos más importantes dados en EF 4.0 está precisamente en las posibilidades de adaptación/personalización de EF. Así, tendremos la posibilidad de decidir cómo queremos implementar nuestras ENTIDADES del Dominio (ligadas a EF, POCO, IPOCO, Self-Tracking, etc.), manejar las asociaciones entre los mismos, e implementar un patrón Repository para trabajar contra el API.

5.4.- Creación del Modelo de Datos Entidad-Relación de Entity-Framework

Primeramente debemos resaltar que esta guía no pretende enseñar 'paso a paso' (tipo *Walkthrough*) como utilizar Visual Studio ni .NET 4.0, de eso se encarga un gran volumen de documentación de Microsoft o libros relacionados. Así pues tampoco explicaremos absolutamente todos los 'por menores'. En cambio, si pretendemos mostrar el mapeo entre la tecnología y una Arquitectura N-Capas con Orientación al Dominio.

Sin embargo, en lo relativo a las entidades POCO/IPOCO en EF 4.0, sí lo haremos paso a paso, por ser algo bastante nuevo en VS y EF.

Para crear el modelo, partiremos de un proyecto de tipo librería de clases (típica .DLL). Este ensamblado/proyecto contendrá todo lo relacionado con el modelo de datos y conexión/acceso a la base de datos, para un módulo funcional concreto de nuestra aplicación.

En nuestro ejemplo, el proyecto lo llamamos así (coincide con el *NameSpace*):

`"Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule"`

Nótese que en este caso, al módulo vertical/funcional le llamamos simplemente 'MainModule'. Podremos tener otros módulos denominados 'RRHH', 'CRM', o cualquier otro concepto funcional.

A dicho proyecto/ensamblado, le añadimos un modelo de datos de EF, llamado en nuestro ejemplo '**MainModuleDataModel**':

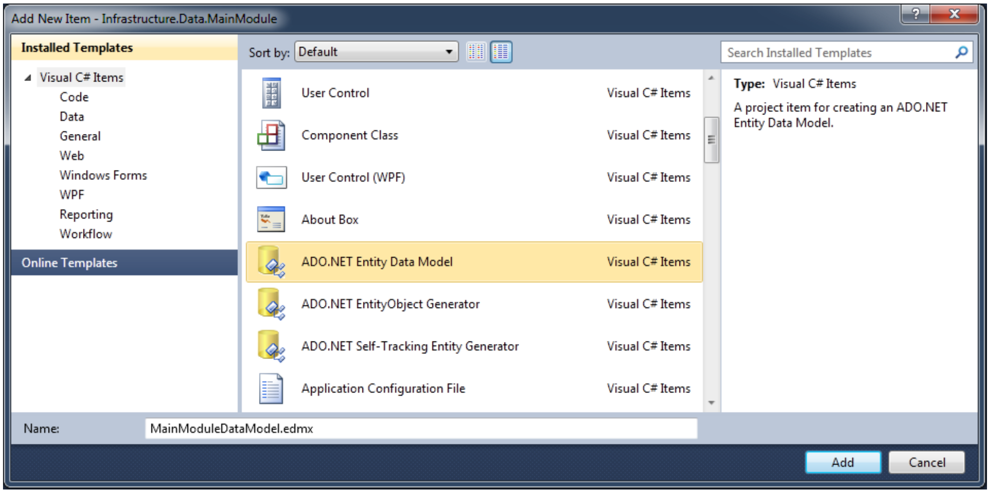


Figura 4.- Creación Modelo Entidades de Datos de EF

Si el modelo lo vamos a crear a partir de una base de datos, se nos pedirá cual es dicha base de datos. Es muy importante denominar correctamente el nombre con que va a guardar el **string de conexión**, porque realmente ese nombre será el mismo que el de la clase de Contexto de EF de nuestro módulo. Así, en nuestro ejemplo, lo llamamos **MainModuleContext** (Contexto de nuestro Módulo Funcional principal de la aplicación)**:**

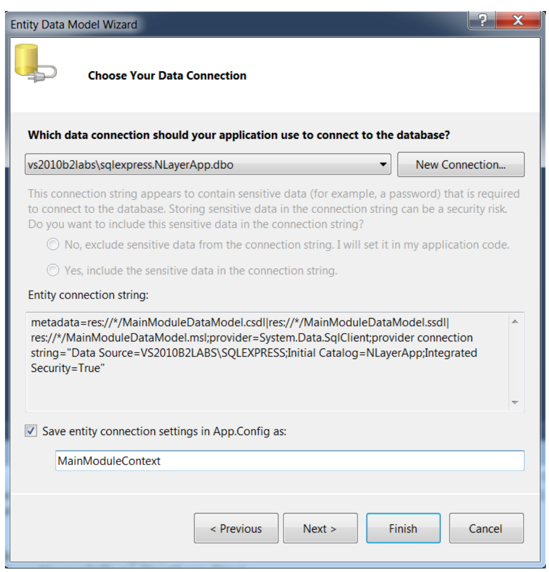


Figura 5.- Asistente para MainModuleContext

Al añadir las tablas (o crear el modelo nuevo de cero), debemos seleccionar un nombre del *namespace* relacionado con nuestro módulo vertical/funcional, por ejemplo, **NLayerApp.MainModule**. Este namespace no es de código .NET. Es un namespace interno del modelo de EF (dentro del .edmx).

Aquí es también importante comprobar que incluimos las columnas de '*foreign key*' (claves extranjeras) y en caso de que nuestras tablas estén denominadas en inglés en singular, es útil decir que a los nombres de los objetos generados los pluralice o singularice, teniendo en cuenta que esta pluralización solo funciona bien con entidades que tengan nombres en inglés. A continuación se muestra este paso:

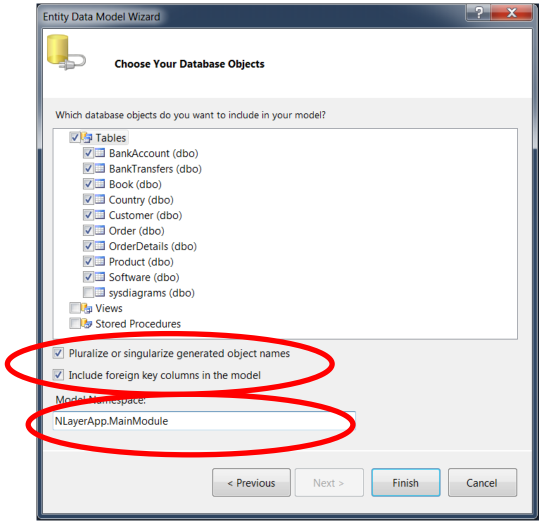


Figura 6.- NameSpace del Modelo EF: NLayerApp.MainModule

Así finalmente podemos disponer del siguiente modelo (coincide con el modelo de datos de nuestra aplicación ejemplo de la Arquitectura):

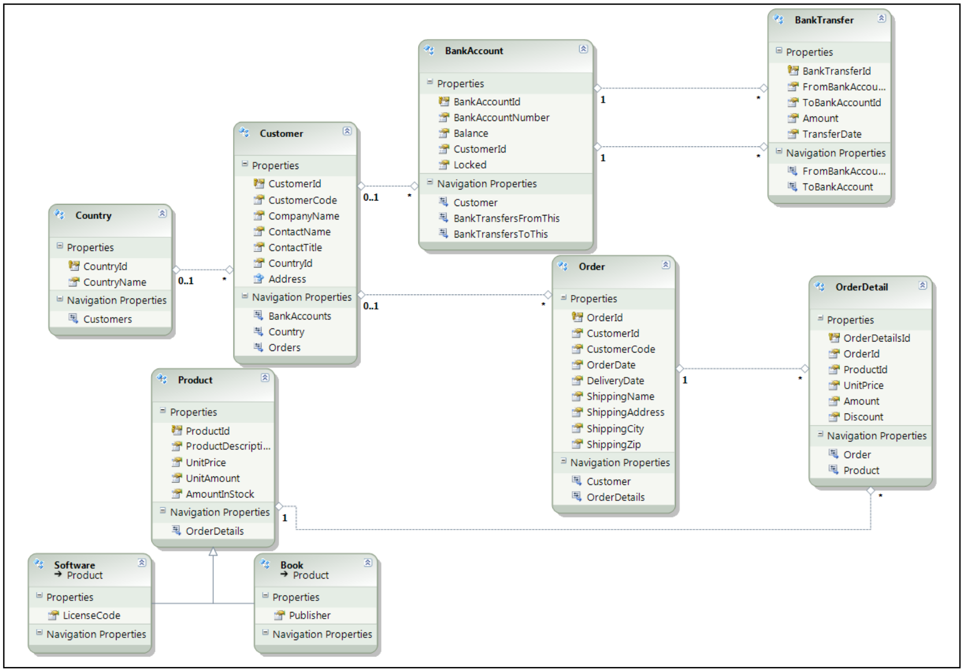


Figura 7.- Modelo de Entidades del Dominio

Y en la vista '*Model Browser*' lo veríamos así:

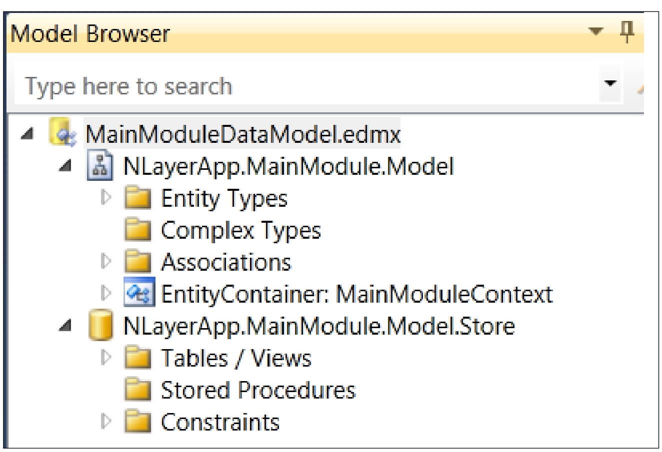


Figura 8.- Vista 'Model Browser'

5.5.- Plantillas T4 de generación de entidades POCO/Self-Tracking

En *Visual Studio 2010* se nos proporcionan plantillas T4 de generación de código que nos generan entidades POCO o *Self-Tracking* (IPOCO) a partir del modelo de datos entidad-relación de *Entity Framework* en nuestro proyecto.

Deberemos tener normalmente un modelo de datos EDM de Entity Framework por cada módulo funcional de nuestra aplicación, aunque por supuesto, esto son decisiones de diseño, depende también del volumen de entidades de cada módulo, etc.

La decisión de cuándo hacer uso de entidades POCO versus *Self-Tracking* (IPOCO) está descrita en el capítulo de 'Capa del Dominio'. **En el ejemplo de la Arquitectura Marco hemos decidido hacer uso de entidades *Self-Tracking* (IPOCO) por ser mucho más potentes para una aplicación N-Tier (proporcionan gestión automática de concurrencia optimista) y adicionalmente requerir mucho menos coste de desarrollo que las entidades puramente POCO si se quiere conseguir el mismo nivel de funcionalidad**.

T4 es una herramienta de generación de código incluida en Visual Studio. Las plantillas de T4 pueden ser modificadas por nosotros para producir diferentes patrones de código basados en ciertas premisas de entrada.

Añadir plantillas T4

Desde cualquier punto en blanco de nuestro diseñador EDM de EF, se debe hacer clic con el botón derecho y seleccionar "**Add Code Generation Item…**", con un menú similar al siguiente:

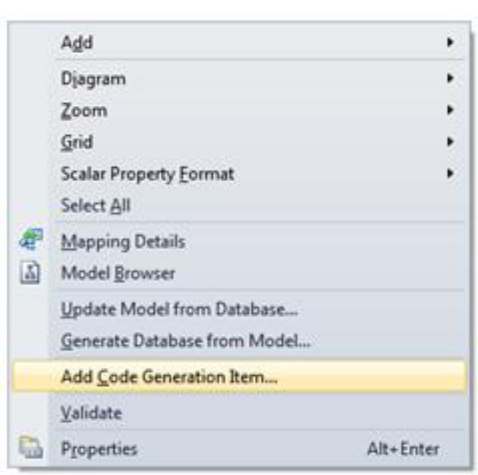


Figura 9.- Add Code Generation Item…

Esto muestra un diálogo “**Add New Item**”. Seleccionamos el tipo “**ADO.NET Self-Tracking Entity Generator**” y especificamos, por ejemplo, “**MainModuleModel.tt**”:

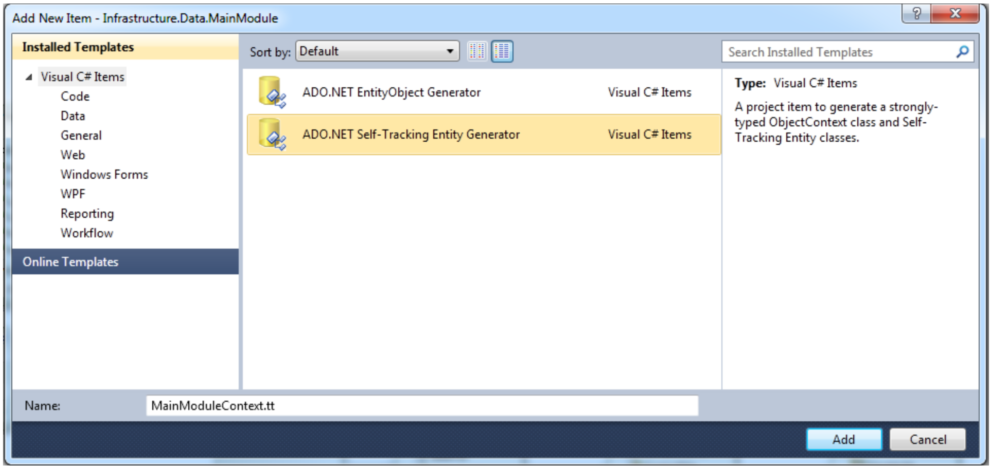


Figura 10.- Creación Plantillas T4 para Entidades ‘Self-Tracking’

Este paso realmente no ha generado un único fichero T4 con el nombre que proporcionamos, sino que lo que ha generado son dos ficheros plantilla T4. El primero de ellos nos sirve para generar las propias clases de Entidades o tipos de datos (en este caso con el nombre **MainModuleModel.Types.tt** y serán IPOCO de tipo *Self-Tracking*), y el segundo fichero T4 es el que genera las clases con conexión a la infraestructura de *Entity Framework* (en este caso con el nombre **MainModuleModel.Context.tt**).

Esto lo que básicamente ha ocasionado es que se deshabilita en Visual Studio la generación normal de clases entidad ligadas a Entity Framework (con dependencia directa de infraestructura), y a partir de ahora, son nuestras plantillas T4 quienes serán las encargadas de generar dichas clases pero ahora de tipo *Self-Tracking* o POCO.

Internamente, en la platilla T4, se le está especificando el *path* al fichero del modelo de *Entity Framework*. En este caso, si se abre cualquiera de las dos platillas TT, se verá una línea donde se especifica algo así:

```
string inputFile = @"MainModuleDataModel.edmx";
```

Siempre que se modifiquen estas plantillas T4, al grabarse, se volverán a generar las clases entidad, contexto, etc.

En este momento debemos tener algo similar a lo siguiente:

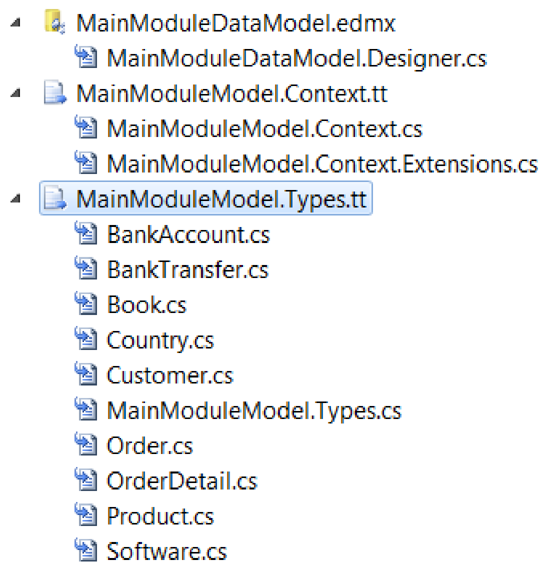


Figura 11.- Plantillas TT y clases generadas

Si se realizan modificaciones en el modelo y queremos propagarlas a las clases entidad, solo hay que seleccionar la opción '*Run Custom Tool*' del menú botón derecho sobre los ficheros .tt, así:

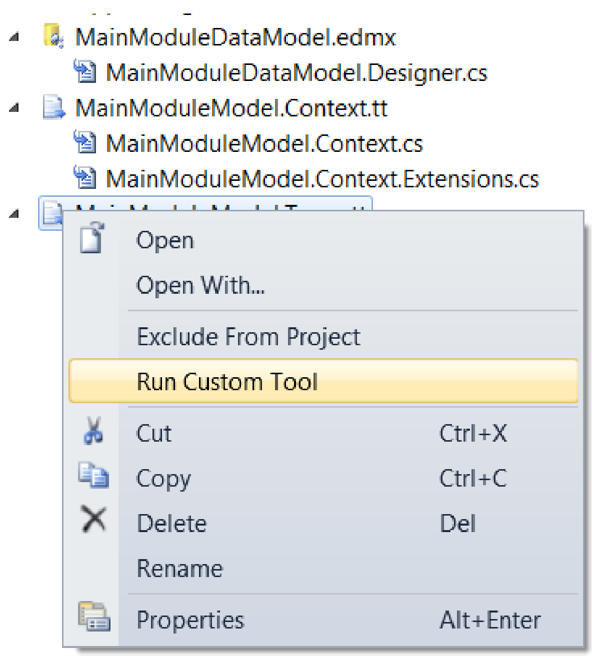


Figura 12.- Run Custom Tool

5.6.- Tipos de datos ‘Entidades Self-Tracking’

Aunque el código generado para las entidades *Self-Tracking Entities (STE)* y POCO es similar al utilizado para las entidades normales de EF (ligadas a clases base de EF), ahora se aprovecha el nuevo soporte al principio PI (*Persistance Ignorance*) disponible en EF 4.0. Así pues, el código generado por las plantillas T4 para nuestras entidades STE no contiene atributos o tipos definidos directamente en EF. Gracias a esto, las entidades *self-tracking* y POCO pueden ser utilizadas también en Silverlight sin ningún problema.

Se puede estudiar el código generado en cualquiera de las clases generadas (p.e. en nuestro caso ejemplo, “Customer.cs”):

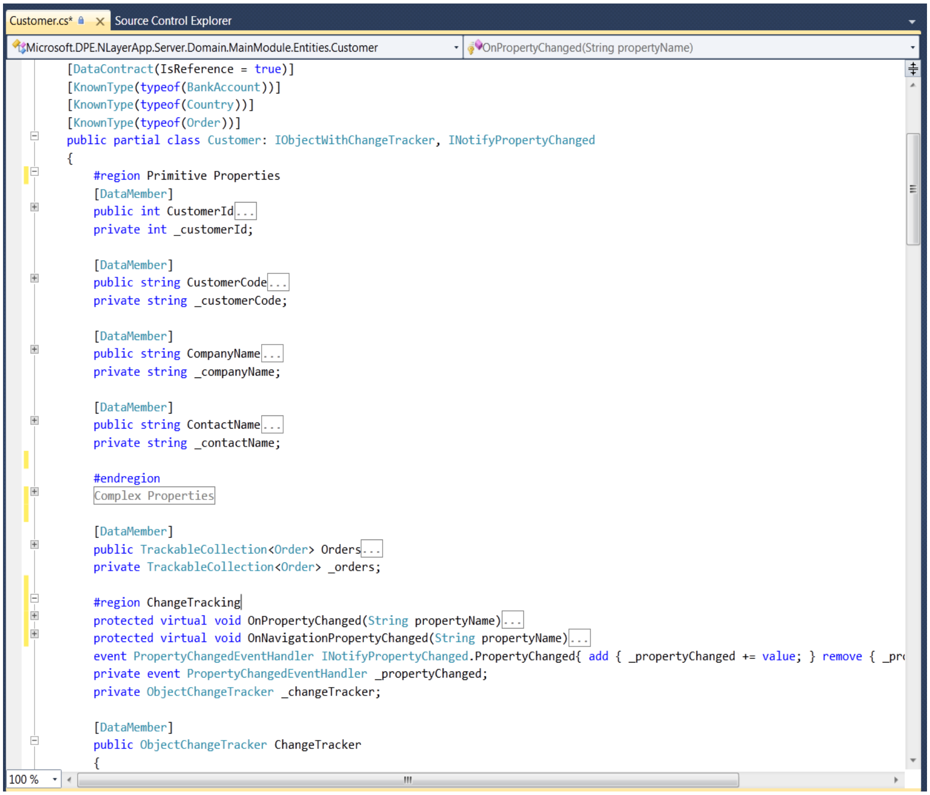


Figura 13.- Ejemplo de Clase Entidad Customer.cs

Puntos a destacar en una entidad '*Self-Tracking*':

1.- Hay un atributo de tipo '**DataContract**' con la propiedad `IsReference = true` en cada tipo de entidad y todas las propiedades públicas están marcadas como **DataMember**. Esto permite a WCF (para las comunicaciones remotas) el serializar grafos bidireccionales con ciclos.

2.- **TrackableCollection** es un tipo de colección basada en **ObservableCollection** que también se incluye en el código generado y tiene la capacidad de notificar cada cambio individual producido en la colección (a su vez deriva de **ObservableCollection** de .NET Framework). Las entidades Self-Tracking usan este tipo para las propiedades de navegación de colecciones. La notificación se utiliza para propósitos de seguimiento de cambios pero también para alinear varios elementos que representan la misma relación cuando uno de ellos cambia. Por ejemplo, cuando un 'Pedido' se añade a la colección de Pedidos de 'Cliente', la referencia al dueño de 'Pedido' se actualiza también para que apunte al 'Cliente' correcto y la propiedad de clave extranjera OwnerID se actualiza con el ID del dueño.

3.- La propiedad **ChangeTracker** proporciona acceso a la clase **ObjectChangeTracker** que almacena y controla la información de seguimiento de cambios de la entidad en cuestión. Esto se utilizará internamente cuando hagamos uso de Gestión de excepciones de Concurrencia Optimista.

Para hacer posible el obtener instancias de entidades 'self-tracking' en el lado cliente, tendremos que compartir el código de los propios tipos (al final, la DLL donde estén definidas las entidades), entre las capas del servidor y también del cliente (no se hará un simple 'Add Service Reference', también se compartirán los tipos). **Por eso mismo, las entidades 'self-tracking' son adecuadas para aplicaciones N-Tier ya que controlamos su desarrollo extremo a extremo. No son en cambio adecuadas para aplicaciones en las que no se quieren compartir los tipos de datos reales entre el cliente y el servidor, por ejemplo, aplicaciones puras SOA en las que controlamos solo uno de los extremos, bien el servicio o el consumidor. En estos otros casos en los que no se puede ni debe compartir tipos de datos (como SOA puro, etc.), se recomienda hacer uso de DTOs propios (Data Transfer Objects). Este punto está más extendido en el capítulo de Servicios Distribuidos.**

5.7.- Importancia de situar las Entidades en la Capa del Dominio

Debido a lo explicado en capítulos anteriores sobre la independencia del Dominio con respecto a aspectos de tecnología e infraestructura (conceptos en DDD), es

importante situar las entidades como elementos de la 'Capa de Dominio'. Son al fin y al cabo, 'Entidades del Dominio'. Para esto, debemos mover el código generado (T4 y sub-ficheros, en este caso ejemplo **MainModuleDataModel.Types.tt**) al proyecto destinado a hospedar a las entidades del dominio, en este caso, el proyecto llamado en nuestro ejemplo:

'**Microsoft.Samples.NLayerApp.Domain.MainModule.Entities**'

Otra opción, en lugar de mover físicamente los ficheros, sería crear un link o hipervínculo de Visual Studio a dichos ficheros. Es decir, podríamos seguir situando los ficheros físicos en el proyecto de *DataModel* donde se crearon por Visual Studio, pero crear enlaces/links desde el proyecto de entidades. Esto ocasionará que los clases entidad reales se generen dónde queremos, es decir, en **el *assembly* de entidades del dominio** '**Microsoft.Samples.NLayerApp.Domain.MainModule.Entities**' y sin necesidad de mover físicamente los ficheros de la situación física en que los situó Visual Studio y el asistente de EF, y sin necesidad de editar el fichero de la plantilla para que especifique un path relativo al fichero EDMX. Sin embargo, esta forma (links/enlaces), ocasiona algunos problemas, por lo que optamos por mover físicamente la plantilla T4 de las entidades al *assembly* '**Domain.MainModule.Entities**' (Assembly de entidades del Dominio).

Lo primero que debemos hacer es 'limpiar' el T4 que vamos a mover. Para ello, primero deshabilitamos la generación de código de la plantilla T4 **MainModuleDataModel.Types.tt.** Seleccionamos el fichero en el '*Solution Explorer*' y vemos sus propiedades. Tenemos que eliminar el valor de la propiedad '**Custom Tool**' y dejarlo en blanco.

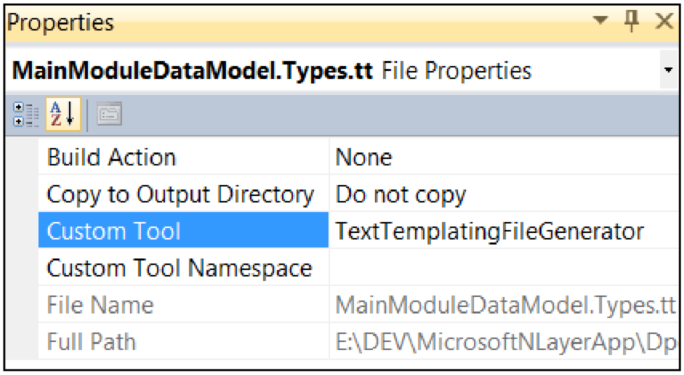


Figura 14.- Custom Tool

También, los ficheros que cuelgan de la plantilla (ficheros .cs de las clases generadas), debemos eliminarlos/borrarlos, porque a partir de ahora no se deben generar en este proyecto:

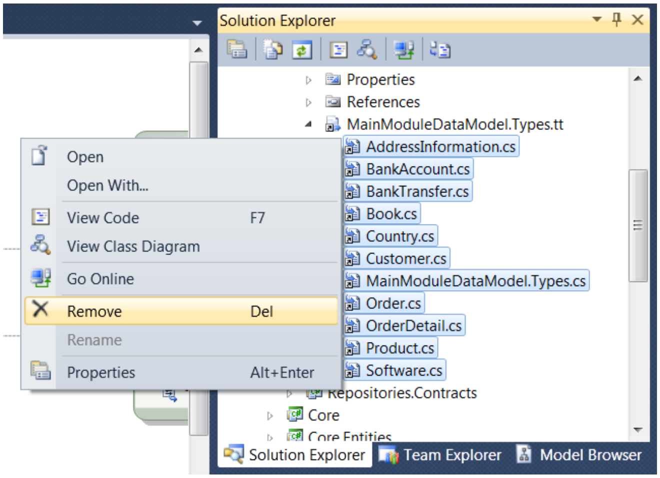


Figura 15.- Eliminar ficheros .cs de las clases entidad generadas

Así pues, simplemente, el fichero "MainModuleModel.tt" tenemos que excluirlo de su proyecto actual (assembly de la capa de persistencia con el modelo EDMX de EF), y copiar físicamente este fichero TT a la carpeta de un nuevo *assembly* (dentro de la Capa de Dominio) que hayamos creado para contener exclusivamente las entidades del dominio. En nuestro caso y en la aplicación ejemplo, en el proyecto "**Domain.MainModule.Entities**". Lógicamente, después de copiarlo, lo debemos añadir como parte del proyecto de Visual Studio.

IMPORTANTE: Una vez copiado el fichero TT al nuevo proyecto de entidades de domino, debemos modificar en la plantilla TT el path al modelo de entidades (.EDMX). Así, por lo tanto, la línea del path nos quedará similar a la siguiente:

```
//(CDLTLL) Changed path to edmx file correct location
string inputFile =
@"..\Infrastructure.Data.MainModule\Model\MainModuleDataModel.edmx";
```

Y finalmente, teniendo ya el fichero T4 (TT) de entidades en su proyecto definitivo y modificado el *path* para que apunte al modelo .EDMX de EF, podemos proceder a probar y generar las clases reales de entidades, haciendo clic con el botón derecho y seleccionando la opción '*Run Custom Tool*':

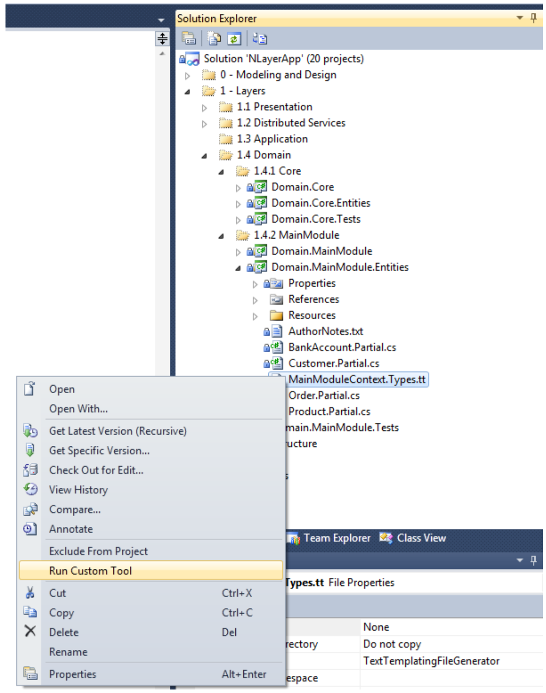


Figura 16.- Generar Clases Entidad con 'Run Custom Tool'

Esto nos generará todas las clases de entidades con el *namespace* correcto (*namespace* de *assembly* del Dominio), etc.:

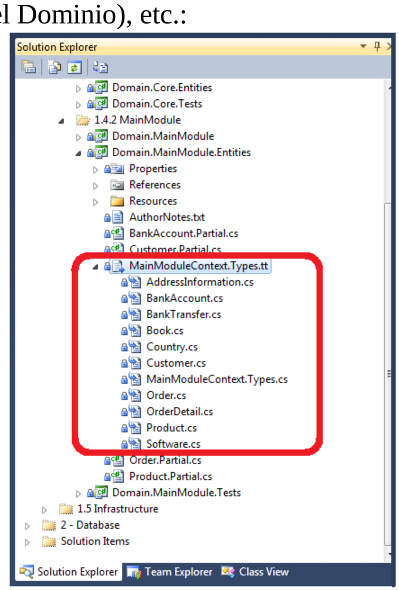


Figura 17.- Clases Entidad en el Dominio

Estas clases, son por lo tanto, código generado y no debemos modificarlas directamente en sus ficheros de clase pues la siguiente vez que Visual Studio genere el código de esas entidades gracias a la plantilla T4, el código que hubiéramos escrito ahí directamente, lo perderíamos.

Sin embargo, y como veremos en el capítulo de Capa del Modelo de Dominio, siempre deberemos añadir lógica del Dominio a las clases entidad mediante clases '*partial*' que podemos añadir.

5.7.1.- Separación del 'Core' de plantillas T4 STE

Con la generación de plantillas T4 STE de VS.2010 se nos generan dos plantillas, una para las entidades desconectadas y otra para los objetos que tienen conexión contra la base de datos (contexto, etc.) En nuestra arquitectura de aplicación normalmente podremos tener varios módulos, y cada uno de ellos deberá disponer de su propio modelo de entidades (sus plantillas T4). Pero, de dichas plantillas generadas para cada módulo, hay una parte común ('Core') que es mejor extraer a una tercera plantilla T4 de forma que no tengamos código redundante en los diferentes módulos. A dicha plantilla la hemos llamado '**ObjectChangeTracker.Types.tt**' y es el código encargado de realizar el 'tracking' (seguimiento) de cambios de las entidades.

Así pues, al estar dentro de *Domain.Core.Entities*, será un código reutilizado desde los diferentes módulos (p.e. desde el módulo *Domain.MainModule.Entities* y otros módulos adicionales que existieran). No hay necesidad de duplicar dicho código en cada módulo y modelo de datos.

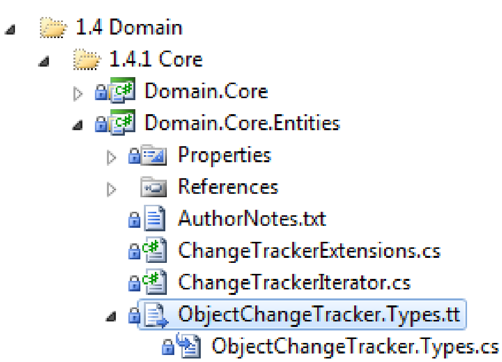


Figura 18.- Plantilla 'Core' ObjectChangeTracker.Types.tt

También es necesario disponer de este código aislado en un assembly diferente porque necesitaremos hacer referencia a él desde los Agentes cliente (WPF, Silverlight, etc.) y poder usar las STE en la capa de presentación. Este último caso es solamente si se ha decidido propagar las entidades del dominio a la capa de presentación haciendo uso de las STE. Si por el contrario se decide hacer uso de DTOs para la capa de

presentación y entidades del dominio solo en la capa de dominio y aplicación, entonces, lógicamente, no se hará referencia a este *assembly* desde el cliente.

Por último también hemos añadido en nuestra aplicación ejemplo algunas extensiones e *Iteradores* implementados en los ficheros '**ChangeTrackerExtension.cs**' y '**ChangeTrackerIterator.cs**'.

5.8.- Plantillas T4 de Persistencia de Datos y conexión a las fuentes de datos

Simultáneamente a la generación de la plantilla TT para las entidades que realizamos antes, también se nos ha generado una plantilla TT para realizar la propia persistencia de datos en la base de datos, llamada en nuestro ejemplo '**MainModuleModel.Context.tt**', es decir, una serie de clases de contexto y con conexión a la base de datos, por lo que son clases completamente ligadas a *Entity Framework*. Precisamente por eso, debe de estar en una capa/subcapa perteneciente a la '*Capa de Infraestructura de Persistencia de Datos*'.

En nuestro ejemplo, lo dejamos situado en el proyecto original 'Microsoft.Samples.NLayerApp.**Infrastructure.Data.MainModule**', si bien, también es factible moverlo a otro proyecto diferente al del modelo .edmx, tal y como hicimos con la plantilla .tt de las entidades.

Las clases de Contexto generadas por esta plantilla TT serán las que utilizaremos posteriormente al desarrollar nuestras clases REPOSITORIO de persistencia y acceso a datos.

5.9.- Implementación de Repositorios con Entity Framework y Linq to Entities

Como se expuso en el capítulo de diseño de esta capa, estos componentes son en algunos aspectos algo similares a los componentes de 'Acceso a Datos' (DAL) de Arquitecturas tradicionales *N-Layered*. Básicamente son clases/componentes que encapsulan la lógica requerida para acceder a las fuentes de datos requeridas por la aplicación. Centralizan por lo tanto funcionalidad común de acceso a datos de forma que la aplicación pueda disponer de un mejor mantenimiento y desacoplamiento entre la tecnología con respecto a la lógica del Dominio. Si se hace uso de tecnologías base tipo O/RM (*Object/Relational Mapping frameworks*) como vamos a hacer con ENTITY FRAMEWORK, se simplifica mucho el código a implementar y el desarrollo se puede focalizar exclusivamente en los accesos a datos y no tanto en la tecnología de acceso a datos (conexiones a bases de datos, sentencias SQL, etc.) que se hace mucho más transparente en ENTITY FRAMEWORK.

Un Repositorio "registra" en memoria (un contexto del almacén) los datos con los que está trabajando e incluso las operaciones que se quieren hacer contra el almacén (normalmente base de datos), pero estas no se realizarán hasta que desde la capa de Aplicación se quieran efectuar esas 'n' operaciones de persistencia/acceso en una misma acción, todas a la vez. Esta decisión de 'Aplicar Cambios' que están en memoria sobre el almacén real con persistencia, está basado normalmente en el patrón 'Unidad de Trabajo' o '*Unit of Work*', definido y utilizado en la Capa de Aplicación.

Como regla general para aplicaciones N-Layer DDD, implementaremos los Repositorios con Entity Framework.

Tabla 7.- Guía de Arquitectura Marco

Regla Nº: I2.	*Implementar Repositorios y clases base con Entity Framework.*

- **Norma**
  - Es importante localizar en puntos bien conocidos (Repositorios) toda la lógica de persistencia y acceso a datos. Deberá existir un Repositorio por cada Entidad raíz del Dominio (Ya sean ENTIDADES sencillas o AGGREGATES). Por regla general y para nuestra Arquitectura Marco, implementaremos los repositorios con Entity Framework.

Referencias

Using Repository and Unit of Work patterns with Entity Framework 4.0
http://blogs.msdn.com/adonet/archive/2009/06/16/using-repository-and-unit-of-work-patterns-with-entity-framework-4-0.aspx

5.10.-Implementación de Patrón Repositorio

Como también expusimos en el capítulo de diseño, un *Repository* es una de las formas bien documentadas de trabajar con una fuente de datos. Otra vez, *Martin Fowler* en su libro PoEAA describe un repositorio de la siguiente forma:

"*Un repositorio realiza las tareas de intermediario entre las capas de modelo de dominio y mapeo de datos, actuando de forma similar a una colección en memoria de*

objetos del dominio. Los objetos cliente construyen de forma declarativa consultas y las envían a los repositorios para que las satisfagan. Conceptualmente, un repositorio encapsula a un conjunto de objetos almacenados en la base de datos y las operaciones que sobre ellos pueden realizarse, proveyendo de una forma más cercana a la orientación a objetos de la vista de la capa de persistencia. Los repositorios, también soportan el objetivo de separar claramente y en una dirección la dependencia entre el dominio de trabajo y el mapeo o asignación de los datos".

Este patrón, es uno de los más habituales hoy en día, sobre todo si pensamos en *Domain Driven Design*, puesto que nos permite de una forma sencilla, hacer que nuestras capas de datos sean *testables* y trabajar de una forma más simétrica a la orientación a objetos con nuestros modelos relaciones .

Así pues, para cada tipo de objeto lógico que necesite acceso global, se debe crear un objeto (Repositorio) que proporcione la apariencia de una colección en memoria de todos los objetos de ese tipo. Se debe establecer el acceso mediante un interfaz bien conocido, proporcionar métodos para añadir y eliminar objetos, que realmente encapsularán la inserción o eliminación de datos en el almacén de datos. Proporcionar métodos que seleccionen objetos basándose en ciertos criterios de selección y devuelvan objetos o colecciones de objetos instanciados (entidades del dominio) con los valores de dicho criterio, de forma que encapsule el almacén real (base de datos, etc.) y la tecnología base de consulta.

Se deben definir REPOSITORIOS solo para las entidades lógicas principales (En un Modelo de Dominio ENTIDADES simples ó *AGGREGATES roots*), no para cualquier tabla de la fuente de datos.

Todo esto hace que se mantenga focalizado el desarrollo en el modelo y se delega todo el acceso y persistencia de objetos a los REPOSITORIOS.

Así pues, a nivel de implementación, un repositorio es simplemente una clase con código de acceso a datos, como puede ser la siguiente clase simple:

C#

```
public class CustomerRepository
{
   …
    // Métodos de Persistencia y acceso a datos
   …
}
```

Hasta aquí no hay nada de especial en esta clase. Será una clase normal e implementaremos métodos del tipo "`Customer GetCustomerById (int customerId)`" haciendo uso de 'Linq to Entities' y como tipos de datos, las propias entidades POCO o *SelfTracking* generadas por EF.

Relativo a esto, deberemos situar los métodos de persistencia y acceso a datos en los Repositorios adecuados, normalmente guiándonos por el tipo de dato o entidad que devolverá un método, es decir, siguiendo la regla expuesta a continuación:

Tabla 8.- Guía de Arquitectura Marco

Regla Nº: I3.	*Situar los métodos en las clases Repositorio dependiendo del tipo de entidad que retornen o actualicen dichos métodos.*

○ **Norma**

- Si un método concreto, definido por ejemplo con la frase "Obtener Clientes de Empresa" devuelve un tipo de entidad concreta (en este caso Customer), el método deberá situarse en la clase de repositorio relacionada con dicho tipo/entidad (en este caso CustomerRepository. No sería en CompanyRepository).

- En caso de estar tratando con sub-entidades dentro de un AGGREGATE, deberemos situar el método en el Repositorio de la clase entidad raíz. Por ejemplo, en el caso de querer devolver todas las líneas de detalle de un pedido, deberemos situar ese método en el Repositorio de la clase entidad raíz del agregado, que es 'OrderRepository'.

- En métodos de actualizaciones, se seguirá la misma regla pero dependiendo de la entidad principal actualizada.

5.10.1.- Clase Base para los Repositories (Patrón 'Layer Supertype')

Antes de ver cómo desarrollar cada uno de sus métodos específicos en .NET y Entity Framework 4.0, vamos a implementar antes una base para todas las clases Repository. Si nos damos cuenta, al final, la mayoría de las clases Repository requieren de un número de métodos muy similar, tipo "ObtenerTodos", "Actualizar", "Borrar", "Nuevo", etc. pero cada uno de ellos para un tipo de entidad diferente. Bien, pues podemos implementar una clase base para todos los Repository (es una implementación del patrón *Layer Supertype* para esta sub-capa de Repositorios) y así poder reutilizar dichos métodos comunes. Sin embargo, si simplemente fuera una clase base y derivamos directamente de ella, el problema es que heredaríamos y utilizaríamos exactamente los mismos métodos de la clase base, con un tipo de datos/entidad concreto. Es decir, algo como lo siguiente no nos valdría:

```
C#

//Clase Base o Layered-Supertype de Repositories
public class GenericRepository
{
     //Métodos base para todos los Repositories
     //Add(), GetAll(), New(), Update(), etc…
}


public class CustomerRepository : Repository
{
   …
    // Métodos específicos de Persistencia y acceso a datos
   …
}
```

Lo anterior no nos valdría, porque al fin y al cabo, los métodos que podríamos reutilizar serían algo que no tengan que ver con ningún tipo concreto de entidad del dominio, puesto que en los métodos de la clase base Repository no podemos hacer uso de una clase entidad concreta como "Products", porque posteriormente puedo querer heredar hacia la clase "CustomerRepository" la cual no tiene que ver inicialmente con "Products".

5.10.2.- Uso de 'Generics' en implementación de clase base Repository

Sin embargo, gracias a las capacidades de *Generics* en .NET, podemos hacer uso de una clase base cuyos tipos de datos a utilizar sean establecidos en el momento de hacer uso de dicha clase base, mediante *generics*. Es decir, lo siguiente si sería muy útil:

```
C#

//Clase Base ó Layered-Supertype de Repositories
public class GenericRepository<TEntity> : where TEntity : class,new()
{
     //Métodos base para todos los Repositories
     //Add(), GetAll(), New(), Update(), etc…
}


public class CustomerRepository : GenericRepository
{
   …
    // Métodos específicos de Persistencia y acceso a datos
   …
}
```

'TEntity' será sustituido por la entidad a usar en cada caso, es decir, "Products", "Customers", etc. De esta forma, podemos implementar una única vez métodos comunes como "`Add(), GetAll(), New(), Update()`" y en cada caso

funcionarán contra una entidad diferente concreta. A continuación exponemos parcialmente la clase base “Repository” que utilizamos en el ejemplo de aplicación N-Layer:

```
C#

//Clase Base ó Layered-Supertype de Repositories
public class GenericRepository<TEntity> : IRepository<TEntity>
    where TEntity : class,IObjectWithChangeTracker, new()
{
    private IQueryableContext _context;

    //Constructor with Dependencies
    public GenericRepository(IQueryableContext context)
    {
        //…
        //set internal values
        _context = context;
    }

    public IContext StoreContext
    {
        get
        {
            return _context as IContext;
        }
    }

    public void Add(TEntity item)
    {
        //…
        //add object to IObjectSet for this type
        (_context.CreateObjectSet<TEntity>()).AddObject(item);
    }

    public void Remove(TEntity item)
    {
        //…

        //Attach object to context and delete this
        // this is valid only if T is a type in model
        (_context).Attach(item);

        //delete object to IObjectSet for this type
        (_context.CreateObjectSet<TEntity>()).DeleteObject(item);

    }

    public void Attach(TEntity item)
    {
        (_context).Attach(item);
    }

    public void Modify(TEntity item)
    {
        //…

        //Set modifed state if change tracker is enabled
        if (item.ChangeTracker != null)
```

```
                item.MarkAsModified();

            //apply changes for item object
            _context.SetChanges(item);
        }

        public void Modify(ICollection<TEntity> items)
        {
            //for each element in collection apply changes
            foreach (TEntity item in items)
            {
                if (item != null)
                    _context.SetChanges(item);
            }
        }

        public IEnumerable<TEntity> GetAll()
        {
            //Create IObjectSet and perform query
            return
(_context.CreateObjectSet<TEntity>()).AsEnumerable<TEntity>();
        }


        public IEnumerable<TEntity> GetBySpec(ISpecification<TEntity>
specification)
        {
            if (specification == (ISpecification<TEntity>)null)
                throw new ArgumentNullException("specification");

            return (_context.CreateObjectSet<TEntity>()
                            .Where(specification.SatisfiedBy())
                            .AsEnumerable<TEntity>());
        }

        public IEnumerable<TEntity> GetPagedElements<S>(int pageIndex,
int pageCount, System.Linq.Expressions.Expression<Func<TEntity, S>>
orderByExpression, bool ascending)
        {
            //checking arguments for this query
            if (pageIndex < 0)
                throw new
ArgumentException(Resources.Messages.exception_InvalidPageIndex,
"pageIndex");

            if (pageCount <= 0)
                throw new
ArgumentException(Resources.Messages.exception_InvalidPageCount,
"pageCount");

            if (orderByExpression == (Expression<Func<TEntity, S>>)null)
                throw new ArgumentNullException("orderByExpression",
Resources.Messages.exception_OrderByExpressionCannotBeNull);

            //Create associated IObjectSet and perform query

            IObjectSet<TEntity> objectSet =
_context.CreateObjectSet<TEntity>();

            return (ascending)
                                ?
```

```
                                          objectSet.OrderBy(orderByExpression)
                                           .Skip(pageIndex * pageCount)
                                           .Take(pageCount)
                                           .ToList()
                                      :

    objectSet.OrderByDescending(orderByExpression)
                                           .Skip(pageIndex * pageCount)
                                           .Take(pageCount)
                                           .ToList();
            }

        }
```

De esta forma hemos definido ciertos métodos comunes (Add(), Delete(), GetAll(), GetPagedElements(), etc.) que podrán ser reutilizados por diferentes *Repositories* de diferentes entidades del dominio. De forma que una clase Repository podría ser así de sencilla inicialmente, sin realmente ninguna implementación directa y sin embargo ya dispondría de implementación real de dichos métodos heredados de la clase base Repository.

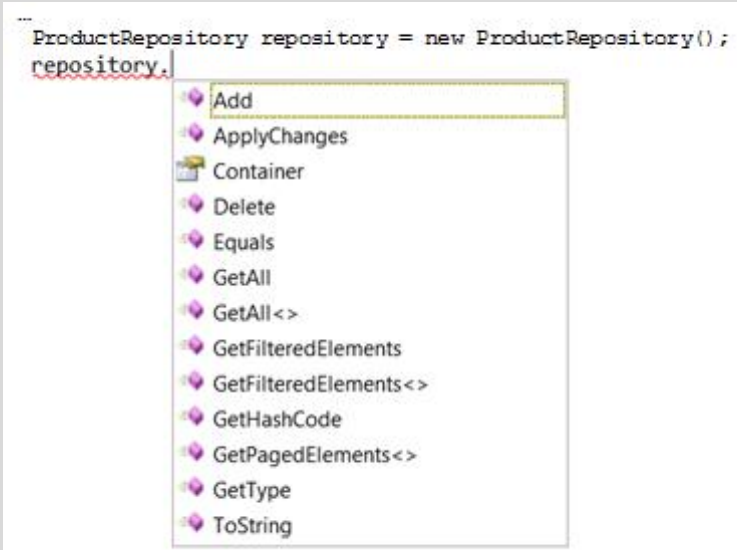
Por ejemplo, así de simple podría ser la implementación inicial de '*ProductRepository*':

```
        C#

        //Clase Repository para entidad Product
        public class ProductRepository : GenericRepository<Product>,
    IProductRepository
            {
                public ProductRepository(IMainModuleContainer container)
                    :base(container)
                {
                }
            }
```

Como se puede apreciar, no hemos implementado ningún método de forma directa en la clase 'ProductRepository', y sin embargo, si instanciamos un objeto de esta clase, estos serían los métodos que "sin hacer nada", ya podríamos utilizar:

C#



Es decir, ya dispondremos de los métodos de consulta, añadir, y borrar básicos y para la entidad concreta de 'Product', sin haberlos implementado específicamente para esta entidad.

Adicionalmente podremos añadir nuevos métodos exclusivos para la entidad 'Product' dentro de la propia clase 'ProductRepository', realizando la implementación con 'Linq to Entities', etc.

La situación de las clases Repositorios, en el ejemplo de aplicación de nuestra Arquitectura, lo tendremos en el siguiente *namespace*, **dentro de la capa de 'Infraestructura de Persistencia de Datos' y para un módulo vertical/funcional concreto (en este caso, el módulo principal denominado *MainModule*)**:

```
Namespace
Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule.Repositories
```

Implementación de métodos concretos en *Repositories* (Adicionales a la clase base)

Un ejemplo de implementación específica de un método de un Repositorio concreto podría ser el siguiente:

C#
```
    //Clase OrderRepository con métodos específicos
    public class OrderRepository
        :GenericRepository<Order>,IOrderRepository
```

```
        {
            public OrderRepository(IMainModuleContext context) :
    base(context) { }


            public IEnumerable<Order> FindOrdersByCustomerCode(string
    customerCode)
            {
                //… Parameters Validations, etc. …

                IMainModuleContext actualContext = base.StoreContext as
    IMainModuleContext;

                //LINQ TO ENTITIES SENTENCE
                return  (from order
                              in actualContext.Orders
                         where
                            order.Customer.CustomerCode == customerCode
                         select
                            order).AsEnumerable();
            }
        }
```

5.10.3.- Interfaces de Repositorios e importancia en el desacoplamiento entre componentes de capas

Aunque hasta ahora hemos introducido simplemente la implementación de las clases de Repositorios, para un correcto diseño desacoplado el uso de abstracciones basadas en Interfaces va a ser fundamental. Por eso, por cada Repositorio que definamos, debemos implementar también su interfaz. Según explicamos en los capítulos teóricos de diseño DDD, serán precisamente estos interfaces lo único que se conocerá desde la capa de Dominio, y la propia instanciación de las clases Repository será realizada por el contenedor IoC elegido (en nuestro caso Unity). De esta forma, tendremos completamente desacoplada la capa de infraestructura de persistencia de datos y sus repositorios de las clases de la capa de Dominio.

Algo importante, sin embargo, es que los interfaces de los Repositorios deben definirse dentro de la Capa de Dominio, puesto que estamos hablando de los contratos que requiere el dominio para que una capa de infraestructura de repositorios pueda ser utilizada de forma desacoplada desde dicho dominio. Así pues, estas abstracciones (interfaces) se definirán en nuestro ejemplo dentro de la capa de **Dominio**, normalmente en una carpeta que agrupará los contratos relacionados con cada Aggregate-Root (Entidad Raiz del Agregado):

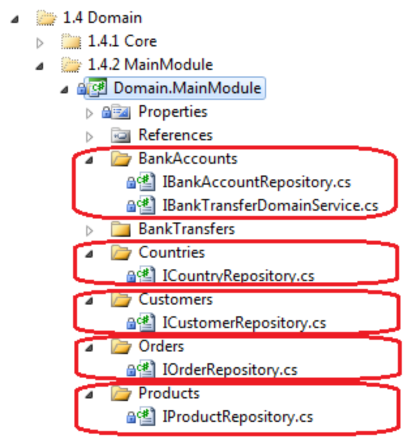


Figura 19.- Las interfaces se definirán dentro de la capa de dominio

De esta forma, mediante abstracciones/interfaces, podríamos llegar a sustituir completamente la capa de infraestructura de persistencia de datos, es decir, los repositorios, sin que esto afectara a las capas de Dominio y Aplicación, ni tener que cambiar dependencias, ni hacer re-compilación alguna. Otra posibilidad, por lo que también es muy importante este desacoplamiento, es el poder hacer *mocking* de los repositorios y que de una forma dinámica las clases de negocio del dominio instancien clases 'falsas' (*stubs* o *mocks*) sin tener que cambiar código ni dependencias, simplemente especificando al contenedor IoC que cuando se le pida que instancie un objeto para un interfaz dado, instancie una clase en lugar de otra (ambas cumpliendo el mismo interfaz, lógicamente). Este sistema de instanciación desacoplada de Repositorios a través de contenedores IoC como Unity, se explica en más detalle en el capítulo de implementación de la Capa de Dominio, pues es ahí donde se deben realizar dichas instanciaciones. Ahora, lo único que es importante a resaltar, es que necesitamos tener definidos interfaces por cada clase Repositorio, y que la situación de dichos interfaces de repositorios estará dentro de la capa de Dominio, por las razones anteriormente mencionadas,

A nivel de implementación de interfaces, el siguiente sería un ejemplo para **ICustomerRepository**:

C#

```
namespace Microsoft.Samples.NLayerApp.Domain.MainModule.Contracts
…
…
//Interfaz/Contrato ICustomerRepository
public interface ICustomerRepository : IRepository<Customer>
{
        Customer GetSingleCustomerByIdWithOrders(int customerId);
        Customer GetSingleCustomerByCustomerCodeWithOrders(string
customerCode);
}
```

Sin embargo, cabe destacar que también en el caso de interfaces de repositorios estamos heredando de un 'interfaz base' (IRepository) que recoge los métodos comunes de los repositorios (add(), delete(), getall(), etc.), y por eso en el anterior interfaz solo estamos declarando otros métodos nuevos/exclusivos del repositorio para 'Customer'.

El interfaz base IRepository sería algo así:

C#

```
namespace Microsoft.Samples.NLayerApp.Domain.Core
…
…
public interface IRepository<TEntity>
        where TEntity : class, new()
    {
        IContainer Container { get; }
        void Add(TEntity item);
        void Delete(TEntity item);
        void Modify(TEntity item);
        void Modify (List<TEntity> items);
        IEnumerable<TEntity> GetAll();
        IEnumerable<K> GetAll<K>() where K : TEntity, new();
        IEnumerable<TEntity> GetPagedElements<S>(int pageIndex, int
pageCount, Expression<Func<TEntity, S>> orderByExpression, bool
ascending = true);
        IEnumerable<TEntity>
GetFilteredElements(Expression<Func<TEntity, bool>> filter);
        …
        …
    }
```

Por lo cual, realmente, a nuestro ICustomerRepository se le suman todos estos métodos heredados.

Como decíamos anteriormente, en este nivel de implementación (Repositorios), simplemente llegamos ahora hasta aquí, pero debemos ser conscientes de cómo vamos a hacer uso de estos repositorios, es decir, haciendo uso de abstracciones (interfaces) e instanciaciones indirectas a través de contenedores IoC, todo esto explicado en el capítulo de implementación de la Capa de Dominio que es donde mayoritariamente se hace uso de los Repositorios.

5.11.- Implementación de Pruebas Unitarias e Integración de Repositorios

La implementación de las pruebas de los repositorios se puede dividir en varios puntos, por un lado la implementación de los elementos comunes en nuestros repositorios, básicamente todos los métodos incluidos en nuestra interfaz *IRepository<TEntity>*, y por otro lado las pruebas de los métodos concretos de cada repositorio.

Para el primer punto, con el fin de ganar productividad en nuestro desarrollo, se ha optado por utilizar herencia de pruebas unitarias, una funcionalidad que ofrecen la mayoría de los frameworks de pruebas y por supuesto también Visual Studio Unit Test.

Para realizar esta tarea, hemos creado la clase base de pruebas *GenericRepositoryTestBase* que implementa todos los métodos genéricos de *GenericRepository* y por lo tanto los métodos comunes para todos los repositorios.

```
C#

[TestClass()]
public abstract class GenericRepositoryTestsBase<TEntity>
        where TEntity : class,IObjectWithChangeTracker, new()
{
            ...
}
```

Algunos ejemplos de pruebas que podemos encontrarnos en esta clase base de test son por ejemplo las siguientes:

```
C#

 [TestMethod()]
 public virtual void AddTest()
 {
     //Arrange
      IQueryableContext context = GetContext();

     //Act
     GenericRepository<TEntity> repository = new
GenericRepository<TEntity>(context);

     TEntity item = new TEntity();
      repository.Add(item);
  }

  [TestMethod()]
  [ExpectedException(typeof(ArgumentNullException))]
  public virtual void AddWithNullTest()
  {
      //Arrange
      IQueryableContext context = GetContext();

      //Act
      GenericRepository<TEntity> repository = new
GenericRepository<TEntity>(context);
      repository.Add(null);

   }
```

Si observa el código anterior, verá cómo se aprovecha la característica de genericidad dentro del lenguaje y como la dependencia de nuestras clases repositorio con la interfaz **IContext** es resuelta mediante un método llamado *GetContext*. La existencia de este método, viene a dar solución a la posibilidad de ejecutar las pruebas de repositorios con un objeto simulado del contexto de trabajo, que en nuestro caso es

Entity Framework, con el fin de conseguir que las pruebas se ejecuten con mayor rapidez y de forma aislada a esta dependencia, externa al fin y al cabo para los repositorios.

```
C#

public IMainModuleContext GetContext(bool initializeContainer = true)
{
    // Get context specified in unity configuration
    // Set active context for
    // testing with fake or real context in application configuration
    // "defaultIoCContainer" setting

     IMainModuleContext context =
IoCFactory.Resolve<IMainModuleContext>();

    return context;

}
```

Tabla 9.- Implementación de pruebas

Regla Nº: I4. **Implementacion de pruebas en los repositorios**

○ **Recomendaciones**

- Disponer de una clase base de test si sus repositorios utilizan un tipo común con una funcionalidad genérica con el fin de ganar productividad a la hora de realizar las pruebas.
- Inyectar las dependencias con un contenedor de dependencias en las pruebas de los repositorios nos permite sustituir las pruebas reales contra una base de datos para realizarlas con algún objeto simulado.

Una vez conseguida nuestra clase base de test, si queremos realizar las pruebas de un determinado repositorio, por ejemplo de **ICustomerRepository**, solamente tenemos que crear una clase de pruebas que herede de **GenericRepositoryBaseTest**.

C#

```
[TestClass()]
 public class CustomerRepositoryTests
     :GenericRepositoryTestsBase<Customer>
 {
 }
```

Es en estas clases donde además se incluyen las pruebas para aquellos métodos concretos de los que disponga el repositorio para el que se estén realizando las pruebas.

C#

```
[TestClass()]
 public class CustomerRepositoryTests
     :GenericRepositoryTestsBase<Customer>
 {
         [TestMethod()]
         [ExpectedException(typeof(ArgumentNullException))]
         public void
FindCustomer_Invoke_NullSpecThrowNewArgumentNullException_Test()
         {
             //Arrange
             IMainModuleContext context = GetContext();
             ICustomerRepository repository = new
CustomerRepository(context);

             //Act
             repository.FindCustomer(null);
         }
     …
     …
 }
```

Para la implementación de las simulaciones, la solución adoptada fue la de la creación de un objeto simulado en un nuevo proyecto, al que se ha llamado como **Infraestructure.Data.MainModule.Mock**. El principal motivo de esta decisión viene dado del hecho de que necesitemos sustituir la dependencia real de los repositorios, con EF, en otras capas de la solución, por lo que este componente podría ser reutilizable.

El mecanismo usado para realizar la simulación de la interfaz *IContext* se basa en la capacidad de Microsoft PEX/MOLES para generar 'stubs' de clases e interfaces de nuestro código. Una vez agregado un archivo de 'moles' dentro del proyecto que albergará nuestro objeto simulado, tendremos disponible un stub de la interfaz ***IContext***, más concretamente **IMainModuleContext** para el caso del módulo principal**.** Aunque podríamos utilizar este *stub* directamente, requeriría de un proceso previo de configuración, asignación de los delegados para especificar los comportamientos, en cada una de sus utilizaciones, por lo que, en esta implementación se decantó por la creación de una clase que heredara del stub creado y especificara completamente sus comportamientos. En el módulo principal esta clase recibe el

nombre de **MainModuleFakeContext** y, a continuación, podemos ver un fragmento de la misma:

```
C#

public class MainModuleFakeContext
        :
Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule.Context.Moles.SIMainModuleContext
    {
        private void InitiateFakeData()
        {

            //configure country
            this.CountriesGet = () => CreateCountryObjectSet();
            this.CreateObjectSet<Entities.Country>(()=>
                CreateCountryObjectSet());
            ...
        }
        ...
        ...

    }
```

Si echa un vistazo al método de inicialización de los datos simulados podrá ver como para cada una de las propiedades *IObjectSet<TEntity>* definidas dentro de la interfaz *IMainModuleContext* debemos especificar un delegado que permita obtener su resultado, al fin y al cabo estos son los elementos consultables por los repositorios y de los cuales puede obtener las colecciones de datos, filtros etc. La creación de objetos de tipo *IObjectSet* es fundamental entonces para la configuración de las simulaciones, por ello, dentro del proyecto **Infraestructure.Data.Core** se dispone de la clase *InMemoryObjectSet*, la cual permite la creación de elementos *IObjectSet* a partir de simples colecciones de objetos.

```
C#

public sealed class InMemoryObjectSet<TEntity> : IObjectSet<TEntity>
        where TEntity : class
    {
        ...

        ...
    }
```

```
C#

IObjectSet<Entities.Country> CreateCountryObjectSet()
        {
            return _Countries.ToInMemoryObjectSet();
        }
```

5.12.- Conexiones a las fuentes de datos

El ser consciente de la existencia de las conexiones a las fuentes de datos (bases de datos en su mayoría) es algo fundamental. Las conexiones a bases de datos son recursos limitados tanto en esta capa de persistencia de datos como en el nivel físico de la fuente de datos. Ténganse en cuenta las siguientes guías, si bien, muchas de ellas son transparentes cuando hacemos uso de un O/RM:

- Abrir las conexiones contra la fuente de datos tan tarde como sea posible y cerrar dichas conexiones lo más pronto posible. Esto asegurará que los recursos limitados se bloqueen durante el tiempo más corto posible y estén disponibles antes para otros consumidores/procesos. Si se hace uso de datos no volátiles, lo más recomendable es hacer uso de concurrencia optimista para incurrir en el coste de bloqueos de datos en la base de datos. Esto evita la sobrecarga de bloqueos de registros, incluyendo también que durante todo ese tiempo también se necesitaría una conexión abierta con la base de datos, y bloqueada desde el punto de vista de otros consumidores de la fuente de datos.

- Realizar transacciones en una única conexión siempre que sea posible. Esto permite que la transacción sea local (mucho más rápida) y no como transacción promovida a distribuida si se hace uso de varias conexiones a bases de datos (transacciones más lentas por la comunicación inter-proceso con el DTC).

- Hacer uso del 'Pooling de conexiones' para maximizar el rendimiento y escalabilidad. Para esto, las credenciales y resto de datos del 'string de conexión' deben de ser los mismos, por lo que no se recomienda hacer uso de seguridad integrada con impersonación de diferentes usuarios accediendo al servidor de bases de datos si se desea el máximo rendimiento y escalabilidad cuando se accede al servidor de la base de datos. Para maximizar el rendimiento y la escalabilidad se recomienda siempre hacer uso de una única identidad de acceso al servidor de base de datos (solo varios tipos de credenciales si se quiere limitar por áreas el acceso a la base de datos). De esa forma, se podrán reutilizar las diferentes conexiones disponibles en el 'Pool de conexiones'.

- Por razones de seguridad, no hacer uso de 'System' o DSN (User Data Source Name) para guardar información de conexiones.

Relativo a la seguridad y el acceso a las fuentes de datos, es importante delimitar cómo van los componentes a autenticar y acceder a la base de datos y cómo serán los requerimientos de autorización. Las siguientes guías pueden ayudar a ello:

- Relativo a SQL Server, por regla general, es preferible hacer uso de autenticación integrada Windows en lugar de autenticación estándar de SQL Server. Normalmente el mejor modelo es autenticación Windows con subsistema confiado (no personalización y acceso con los usuarios de la aplicación, sino, acceso a SQL Server con cuentas especiales/confiadas). La autenticación Windows es más segura porque no requiere especificar 'password' alguna en el string de conexión, entre otras ventajas.

- Si se hace uso de autenticación estándar SQL Server, se debe hacer uso de cuentas específicas (nunca 'sa') con passwords complejos/fuertes, limitándose el permiso de cada cuenta mediante roles de base de datos de SQL Server y ACLs asignados en los ficheros que se usen para guardar los string de conexión, así como cifrar dichos string de conexión en los ficheros de configuración que se estén usando.

- Utilizar cuentas con los mínimos privilegios posibles sobre la base de datos.

- Requerir por programa que los usuarios originales propaguen su información de identidad a las capas de Dominio/Negocio e incluso a la capa de Persistencia y Acceso a Datos para tener un sistema de autorización más granularizado e incluso poder realizar auditorías a nivel de componentes.

- Proteger datos confidenciales mandados por la red hacia o desde el servidor de base de datos. Tener en cuenta que la autenticación Windows protege solo las credenciales, pero no los datos de aplicación. Hacer uso de IPSec o SSL para proteger los datos de la red interna.

5.12.1.- El 'Pool' de Conexiones a fuentes de datos

El '*Connection Pooling*' permite a las aplicaciones el reutilizar una conexión ya establecida contra el servidor de base de datos, o bien crear una nueva conexión y añadirla al pool si no existe una conexión apropiada en el 'pool'. Cuando una aplicación cierra una conexión, se libera al pool, pero la conexión interna permanece abierta. Eso significa que ADO.NET no requiere crear completamente una nueva conexión y abrirla cada vez para cada acceso, lo cual sería un proceso muy costoso. Así pues, una buena reutilización del pool de conexiones reduce los retrasos de acceso al servidor de base de datos y por lo tanto aumenta el rendimiento de la aplicación.

Para que una conexión sea apropiada, tiene que coincidir los siguientes parámetros: Nombre de Servidor, Nombre de Base de datos y credenciales de acceso. En caso de que las credenciales de acceso no coincidan y no exista una conexión similar, se creará una conexión nueva. Es por ello que la reutilización de conexiones cuando la seguridad es Windows y además impersonada/propagada a partir de los usuarios originales, la reutilización de conexiones en el pool es muy baja. Así pues, por regla general (salvo casos que requieran una seguridad específica y el rendimiento y escalabilidad no sea

prioritario), se recomienda un acceso tipo “Sistema Confiado”, es decir, acceso al servidor de bases de datos con solo unos pocos tipos de credenciales. Minimizando el número de credenciales incrementamos la posibilidad de que cuando se solicita una conexión al ‘pool’, ya exista una similar disponible.

El siguiente es un esquema de un ‘Sub-Sistema Confiado’ según se ha explicado:

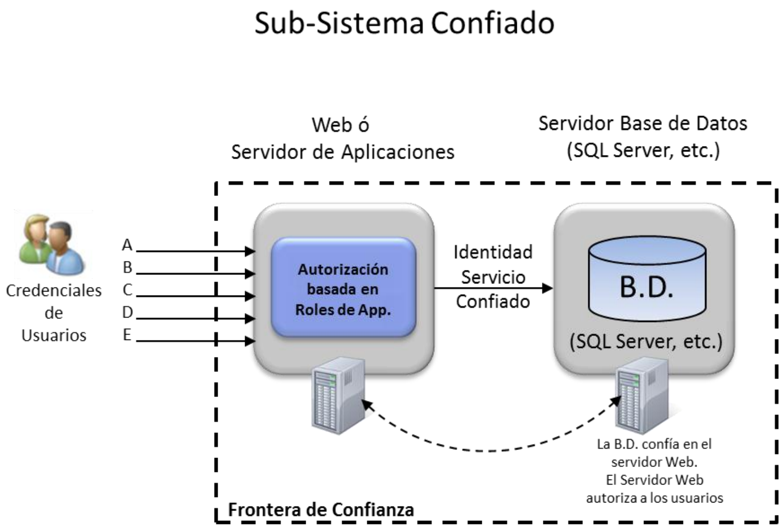


Figura 20.- Esquema de un ‘Sub-Sistema Confiado’

Este modelo de sub-sistema es el más flexible pues permite muchas opciones de control de autorizaciones en el propio servidor de componentes (Servidor de Aplicación), así como realización de auditorías de acceso en el servidor de componentes. Y simultáneamente permite un buen uso del ‘pool de conexiones’ al utilizar cuentas predeterminadas para acceder al servidor de base de datos y poder reutilizar adecuadamente las conexiones disponibles del pool de conexiones.

Finalmente, y completamente al margen, hay objetos de acceso a datos con un rendimiento muy alto (como los DataReaders), pero que sin embargo pueden llegar a ofrecer una mala escalabilidad si no se utilizan correctamente. Esto es así porque es posible que los DataReader mantengan abierta la conexión durante un periodo de tiempo relativamente largo, pues para estar accediendo a los datos requieren tener abierta la conexión. Si hay pocos usuarios, el rendimiento será muy bueno, pero si el número de usuarios concurrentes es muy alto, es posible que empiecen a aparecer problemas de cuellos de botella relacionados con el número de conexiones abiertas simultáneamente y en uso contra la base de datos.

5.13.- Estrategias para gestión de errores originados en fuentes de datos

Es interesante disponer de un sistema homogéneo y una estrategia de gestión de excepciones. Este tema es normalmente un aspecto transversal de la aplicación por lo que debe considerarse el disponer de componentes reutilizables para gestionar las excepciones en todas las capas de una forma homogénea. Estos componentes reutilizables pueden ser componentes/clases propias sencillas o si se tienen requerimientos más complejos (publicación de excepciones en diferentes destinos, como Event Log y traps SNMP, etc.) sería recomendable hacer uso del ***Building Block de Gestión de Excepciones*** de '***Microsoft Enterprise Library***' (v5.0 para .NET 4.0).

Sin embargo, no lo es todo el disponer de una librería o clases reutilizables para implementar la gestión de excepciones en las diferentes capas. Hace falta una estrategia específica a implementar en cada capa. Por ejemplo, hay que tomar las siguientes decisiones:

- Determinar qué tipos de excepciones se propagarán a niveles superiores (normalmente la mayoría) y cuáles serán interceptados y gestionados solo en una capa. En el caso de la capa de 'Infraestructura de Persistencia y Acceso a datos', normalmente deberemos gestionar específicamente aspectos como interbloqueos, problemas de conexiones a la base de datos, algunos aspectos de excepciones de concurrencia optimista, etc.

- Cómo se gestionarán las excepciones que no gestionemos específicamente.

- Considerar el implementar procesos de reintento para operaciones donde se pueden producir '*timeouts*'. Pero hacer esto solo si es factible realmente. Es algo a estudiar caso a caso.

- Diseñar una estrategia apropiada de propagación de excepciones. Por ejemplo, permitir que las excepciones suban a las capas superiores donde serán '*logeadas*' y/o transformadas si es necesario antes de pasarlas al siguiente nivel.

- Diseñar e implementar un sistema de 'logging' y notificación de errores para errores críticos y excepciones que no transmitan información confidencial.

5.14.- Agentes de Servicios Externos (Opcional)

Los ‘Agentes de Servicios’ son objetos que manejan las semánticas específicas de la comunicación con servicios externos (servicios web normalmente), de forma que aíslan a nuestra aplicación de las idiosincrasias de llamar a diferentes servicios y proporcionar servicios adicionales como mapeos básicos entre el formato expuesto por los tipos de datos esperados por los servicios externos y el formato de datos que nosotros utilizamos en nuestra aplicación.

También se pueden implementar aquí sistemas de *cache,* o incluso soporte a escenarios ‘offline’ o con conexiones intermitentes, etc.

En grandes aplicaciones es muchas veces usual que los agentes de servicio actúen como un nivel de abstracción entre nuestra capa de Dominio (Lógica de negocio) y los servicios remotos. Esto puede proporcionar un interfaz homogéneo y consistente sin importar los formatos de datos finales.

En aplicaciones más pequeñas, la capa de presentación puede normalmente acceder a los Agentes de Servicio de una forma directa, sin pasar por los componentes de Capa de Dominio y Capa de Aplicación.

Estos agentes de servicios externos son un tipo de componentes perfectos para tener desacoplados con IoC y poder así simular dichos servicios web con ‘fakes’ para tiempo de desarrollo, y realizar pruebas unitarias de estos agentes.

5.15.- Referencias de tecnologías de acceso a datos

“T4 y generación de código" - ***http://msdn.microsoft.com/en-us/library/bb126445(VS.100).aspx***

N-Tier Applications With Entity Framework - ***http://msdn.microsoft.com/en-us/library/bb896304(VS.100).aspx***

".NET Data Access Architecture Guide" at ***http://msdn.microsoft.com/en-us/library/ms978510.aspx***

"Data Patterns" at ***http://msdn.microsoft.com/en-us/library/ms998446.aspx***

"Designing Data Tier Components and Passing Data Through Tiers" at ***http://msdn.microsoft.com/en-us/library/ms978496.aspx***

CAPÍTULO

5

Capa de Modelo de Dominio

1.- EL DOMINIO

Esta sección describe la arquitectura de las capas de lógica del dominio (reglas de negocio) y contiene guías clave a tener en cuenta al diseñar dichas capas.

Esta capa debe ser responsable de representar conceptos de negocio, información sobre la situación de los procesos de negocio e implementación de las reglas del dominio. También debe contener los estados que reflejan la situación de los procesos de negocio, aun cuando los detalles técnicos de almacenamiento se delegan a las capas inferiores de infraestructura (Repositorios, etc.)

Esta capa, 'Modelo del Dominio', es el corazón del software.

Así pues, estos componentes implementan la funcionalidad principal del sistema y encapsulan toda la lógica de negocio relevante (genéricamente llamado lógica del Dominio según nomenclatura DDD). Básicamente suelen ser clases en el lenguaje seleccionado que implementan la lógica del dominio dentro de sus métodos, aunque también puede ser de naturaleza diferente, como sistemas dinámicos de reglas de negocio, etc.

Siguiendo los patrones de Arquitectura en DDD, esta capa tiene que ignorar completamente los detalles de persistencia de datos. Estas tareas de persistencia deben ser realizadas por las capas de infraestructura.

La principal razón de implementar capas de lógica del dominio (negocio) radica en diferenciar y separar muy claramente entre el comportamiento de las reglas del dominio (reglas de negocio que son responsabilidad del modelo del dominio) de los detalles de implementación de infraestructura (acceso a datos y repositorios concretos ligados a una tecnología específica como pueden ser ORMs, o simplemente librerías de acceso a datos o incluso de aspectos horizontales de la **arquitectura)**. De esta forma (aislando el Dominio de la aplicación) incrementaremos drásticamente la

mantenibilidad de nuestro sistema y podríamos llegar a sustituir las capas inferiores (acceso a datos, ORMs, y bases de datos) sin que el resto de la aplicación se vea afectada.

En la presente sección de la guía, sobre todo se quiere destacar el enfoque en dos niveles. Un primer nivel lógico (Arquitectura lógica, como el presente capítulo), que podría ser implementado con cualquier tecnología y lenguajes (cualquier versión de .NET o incluso otras plataformas no Microsoft) y posteriormente un segundo nivel de implementación de tecnología, donde mostraremos, específicamente con versiones de tecnologías concretas de .NET 4.0, como desarrollar esta capa.

2.- ARQUITECTURA Y DISEÑO LÓGICO DE LA CAPA DE DOMINIO

Esta guía está organizada en categorías que incluyen el diseño de capas de lógica del dominio así como la implementación de funcionalidades propias de esta capa, como **desacoplamiento** con la capa de infraestructura de acceso a datos haciendo uso de **IoC** y **DI**, y conceptos de seguridad, cache, gestión de excepciones, *logging* y validación.

En el siguiente diagrama se muestra cómo encaja típicamente esta capa del modelo de Dominio dentro de nuestra arquitectura '*N-Layer Domain Oriented*':

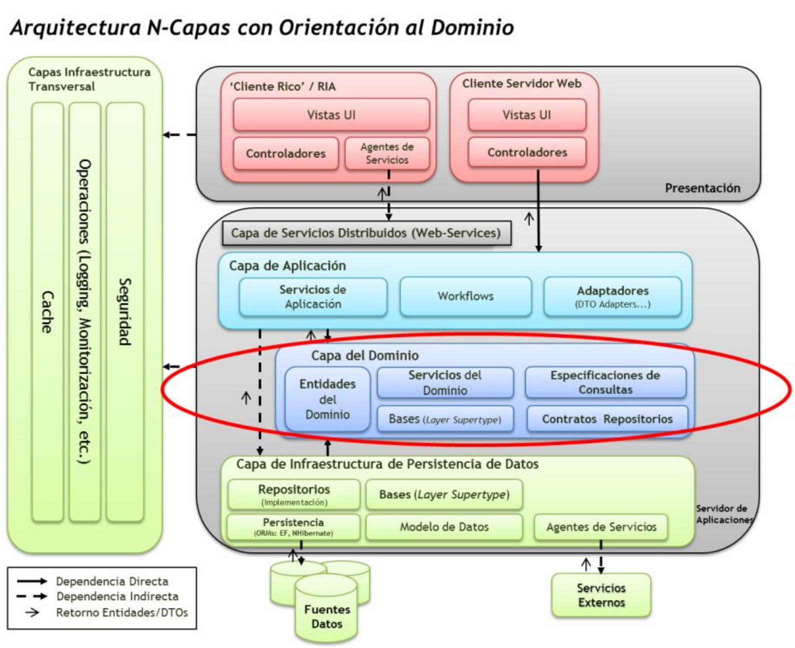


Figura 1.- Arquitectura N-Capas con Orientación al Dominio

2.1.- Aplicación ejemplo: Características de negocio del Modelo de Dominio ejemplo a Diseñar

Antes de continuar con los detalles de cada capa y cómo diseñar cada una internamente siguiendo patrones de diseño orientados al dominio, vamos a exponer un *StoryScript* o '*Modelo de Dominio ejemplo*' que será el que utilicemos para ir diseñando nuestra aplicación en cada capa e incluso para implementarla posteriormente (Aplicación ejemplo).

Nota:
Hemos definido a continuación una lista de requerimientos de negocio muy simplificada. Es de hecho, funcionalmente, extremadamente simple, pero de forma intencionada, especialmente el área relacionada con operaciones bancarias. Esto es así porque el objetivo principal de la aplicación ejemplo es resaltar aspectos de arquitectura y diseño, no de diseñar e implementar una aplicación funcionalmente completa y real.

Los detalles iniciales de requerimientos/problemas del dominio, a nivel funcional, y que habrían sido obtenidos mediante conversaciones con expertos del dominio (usuarios finales expertos en un área concreta funcional), serían los siguientes:

1.- *Se requiere de una aplicación de gestión de* ***clientes y pedidos*** *de dichos clientes. Así mismo, deberá existir otro módulo bancario relacionado con el Banco del grupo para poder realizar* ***transferencias*** *y otras operaciones bancarias de dichos clientes.*

2.- *Lista de Clientes pudiendo aplicar filtros flexibles. Los operadores que gestionan los clientes necesitan poder realizar búsquedas de clientes de una forma flexible. Poder buscar por parte/inicio del nombre y se podría extender en el futuro a búsquedas por otros atributos diferentes (País, Provincia, etc.). También sería muy útil es disponer de búsquedas de clientes cuyos pedidos estén en ciertos estados (p.e. 'impagados'). El resultado de esta funcionalidad requerida es simplemente una lista de clientes con sus datos principales (ID, nombre, localidad, etc.).*

3.- *Lista de Pedidos cuando visualizamos un cliente específico. El valor total de cada pedido deberá estar visible en la lista así como la fecha de pedido y nombre de la persona de referencia.*

4.- *Un pedido puede tener un número indeterminado de líneas de detalle (artículos del pedido).*
Un pedido puede tener muchas líneas de pedido. Cada línea describe un artículo del pedido, que consiste en un producto y el número deseado de unidades de dicho producto.

5.- *Es importante la detección de conflictos de concurrencia.*
Es aceptable el uso de 'Control de Concurrencia Optimista', es decir, es aceptable que cuando un usuario intenta realizar una actualización de datos sobre un conjunto de datos que consultó inicialmente, y mientras estaba trabajando en ell,o (o tomando un café) otro usuario de la aplicación modificó los datos originales en la base de datos, cuando el primer usuario intente actualizar los datos, se detecte este conflicto (datos originales modificados y posibilidad de perderlos al grabar ahora). Solo se deberán considerar los conflictos que ocasionen verdaderas inconsistencias.

6.- *Un pedido no podrá tener un valor total menor de 6 EUROS ni mayor de 1 millón de EUROS.*

7.- *Cada pedido y cada cliente deben disponer de un número/código que sea amigable al usuario, es decir, que sea legible y pueda escribirse y recordarse fácilmente así como la posibilidad de realizar búsquedas por dichos códigos. Si adicionalmente la aplicación gestiona también IDs más complejos, eso debe ser transparente para el usuario.*

8.- *Un pedido tiene que pertenecer a un cliente; una línea de pedido tiene que pertenecer a un pedido. No pueden existir pedidos sin cliente definido. Tampoco pueden existir líneas de pedido que no pertenezcan a un pedido.*

9.- *Las operaciones bancarias podrán ser independientes del módulo de clientes y pedidos. Deberá contemplar una visualización básica de listado de cuentas existentes con sus datos relevantes mostrados en la lista (saldo, número de cuenta, etc.), así como la capacidad de realizar Transferencias bancarias simplificadas entre dichas cuentas (cuenta origen y cuenta destino).*

10.- *La aplicación efectiva de una transferencia bancaria (en este caso persistiendo los cambios oportunos en los saldos de cuenta existentes en la base de datos) debe de realizarse de forma atómica (o todo o nada). Debe ser una transacción atómica.*

11.- *Las cuentas dispondrán de un estado de bloqueo/desbloqueo a nivel de negocio. El gestor de la aplicación deberá poder desbloquear/bloquear cualquier cuenta elegida.*

12.- *Si una cuenta está bloqueada, no se podrán realizar operaciones contra ella (ninguna transferencia ni otro tipo de operaciones). En caso de intentarse cualquier operación contra una cuenta bloqueada, la aplicación deberá*

detectarlo y mostrar una excepción de negocio al usuario de la aplicación, informándole de por qué no se puede realizar dicha operación (porque una cuenta concreta está bloqueada a nivel de negocio).

13.- *(SIMPLIFICACIONES DEL EJEMPLO) Se desea disponer de un ejemplo lo más simple posible a nivel funcional y de diseño de datos, para resaltar especialmente la arquitectura, por lo que debe primar la simplicidad en los datos por encima de diseños normalizados de bases de datos y entidades lógicas. Por ejemplo, el hecho de que un cliente, organización y dirección estén fusionados en la misma entidad lógica e incluso tabla de base de datos, no es en absoluto el mejor diseño, pero en este caso (Aplicación ejemplo) queremos realizar un diseño que maximice la simplificación de diseño funcional de la aplicación. Esta aplicación ejemplo quiere mostrar mejores prácticas en Arquitectura, no en diseño lógico de funcionalidad específica de una aplicación. Así pues, en el mundo irreal de esta aplicación, estas características tienen que tenerse en cuenta a la hora de simplificar el diseño:*

- *Un Cliente/Empresa tendrá una única persona de contacto (Aunque en el mundo real no sea así).*

- *Un Cliente/Empresa tendrá una única dirección (Aunque en el mundo real no sea así y pudiera tener varios edificios/direcciones, etc.)*

En base a estas especificaciones, según avancemos en los diferentes elementos de Arquitectura, iremos identificando elementos concretos de la aplicación ejemplo (entidades concretas, Repositorios concretos, Servicios concretos, etc.)

2.2.- Elementos de la Capa de Dominio

A continuación explicamos brevemente las responsabilidades de cada tipo de elemento propuesto para el Modelo del Dominio:

2.2.1.- Entidades del Dominio

Este concepto representa la implementación del patrón ENTIDADES (*ENTITY pattern*).

Las ENTIDADES representan objetos del dominio y están definidas fundamentalmente por su <u>identidad y continuidad</u> en el tiempo de dicha identidad y no solamente por los atributos que la componen.

Las entidades normalmente tienen una correspondencia directa con los objetos principales de negocio/dominio, como cliente, empleado, pedido, etc. Así pues, lo más normal es que dichas entidades se persistan en bases de datos, pero esto depende

completamente del dominio y de la aplicación. No es una obligación. Pero precisamente el aspecto de 'continuidad' tiene que ver mucho con el almacenamiento en bases de datos. La continuidad significa que una entidad tiene que poder 'sobrevivir' a los ciclos de ejecución de la aplicación. Si bien, cada vez que la aplicación se re-arranca, tiene que ser posible reconstituir en memoria/ejecución estas entidades.

Para diferenciar una entidad de otra, es fundamental entonces el concepto de identidad que las identifica de forma inequívoca incluso aunque dos entidades coincidan con los mismos valores en sus atributos/datos. La identidad en los datos es un aspecto fundamental en las aplicaciones. Un caso de identidad equivocada en una aplicación puede dar lugar a problemas de corrupción de datos y errores de programa. Muchas cosas, en el dominio real (la realidad del negocio) o en el modelo de dominio de la aplicación (abstracción del negocio), están definidas por su identidad y no por sus atributos. Un muy buen ejemplo de entidad es una persona. Los atributos de las entidades pueden ir cambiando a lo largo de su vida, como la dirección, datos financieros e incluso el nombre, y sin embargo, se continúa siendo la misma entidad, la misma persona, en este ejemplo. Por lo tanto, el concepto fundamental de una ENTIDAD es una vida continua abstracta que puede evolucionar a diferentes estados y formas, pero que siempre será la misma entidad.

Algunos objetos no están definidos de forma primaria por sus atributos, representan un hilo de identidad con una vida concreta y a menudo con diferentes representaciones. Una entidad debe poder distinguirse de otra entidad diferente aunque tengan los mismos atributos descriptivos (p.e. pueden existir dos personas con el mismo nombre y apellidos). Los errores de identidad pueden ocasionar corrupción de datos.

Relativo a DDD, y de acuerdo con la definición de Eric Evans, "*A un objeto primariamente definido por su identidad se le denomina ENTIDAD*". Las entidades son muy importantes en el modelo del Dominio y tienen que ser identificadas y diseñadas cuidadosamente. Lo que en algunas aplicaciones puede ser una ENTIDAD, en otras aplicaciones no debe serlo. Por ejemplo, una 'dirección' en algunos sistemas puede no tener una identidad en absoluto, pues puede estar representando solo atributos de una persona o compañía. En otros sistemas, sin embargo, como en una aplicación para una empresa de electricidad, la dirección de los clientes puede ser muy importante y debe ser una identidad porque la facturación puede estar ligada directamente con la dirección. En este caso, una dirección tiene que clasificarse como una ENTIDAD del Dominio. En otros casos, como en un comercio electrónico, la dirección puede ser simplemente un atributo del perfil de una persona. En este otro caso, la dirección no es tan importante y debería clasificarse como un OBJETO-VALOR (En DDD denominado VALUE-OBJECT).

Una ENTIDAD puede ser de muchos tipos, podría ser una persona, un coche, una transacción bancaria, etc. pero lo importante a destacar es que depende del modelo de dominio concreto si es o no una entidad. Un objeto concreto no tiene por qué ser una ENTIDAD en cualquier modelo de dominio de aplicaciones. Así mismo, no todos los objetos en el modelo son ENTIDADES.

Por ejemplo, a nivel de transacciones bancarias, dos ingresos de la misma cantidad y en el mismo día, son sin embargo distintas transacciones bancarias, por lo que tienen

una identidad y son ENTIDADES. Incluso, aun cuando los atributos de ambas entidades (en este caso ingresos) fueran exactamente iguales (incluyendo la hora y minutos exactos), aun así, serían diferentes ENTIDADES. El propósito de los identificadores es precisamente poder asignar identidad a las ENTIDADES.

Diseño de la implementación de Entidades

A nivel de **diseño e implementación**, estos objetos son entidades de datos desconectados y se utilizan para obtener y transferir datos de entidades entre las diferentes capas. Estos datos representan entidades de negocio del mundo real, como productos o pedidos. Las entidades de datos que la aplicación utiliza internamente, son en cambio, estructuras de datos en memoria, como puedan ser clases propias. Si estos objetos entidad son dependientes de la tecnología de acceso a datos (p.e. *Entity Framework 1.0*), entonces estos elementos podrían situarse dentro de la capa de infraestructura de persistencia de datos, puesto que estarían ligados a una tecnología concreta. **Por el contrario, si seguimos los patrones que recomienda DDD y hacemos uso de objetos POCO (*Plain Old CLR Objects*), es decir, de clases independientes, entonces estas ENTIDADES deben situarse mejor como elementos de la capa de Dominio, puesto que son entidades del Dominio e independientes de cualquier tecnología de infraestructura (ORMs, etc.)**.

Tabla 1.- Principio de Desconocimiento de la Tecnología de Persistencia

Principio PI (Persistance Ignorance), POCO e IPOCO
Este concepto, donde se recomienda que la implementación de las entidades del dominio deba ser **POCO (Plain Old Clr Objects)**, es casi lo más importante a tener en cuenta en la implementación de entidades siguiendo una arquitectura orientada al Dominio. Está completamente sustentado en el principio, es decir, que todos los componentes de la Capa de Dominio ignoren completamente las tecnologías con a las que está ligada la Capa de Infraestructura de Persistencia de Datos, como ORMs. Y en concreto, las clases entidad, también deben ser independientes de las tecnologías utilizadas en la Capa de Infraestructura de Persistencia de Datos. Por eso deben ser implementadas como clases POCO (Clases .NET independientes). La forma en cómo estos objetos entidad sean implementados, toma una importancia especial para muchos diseños. Por un lado, para muchos diseños (como en DDD) es vital aislar a estos elementos de conocimiento alguno de tecnologías base de acceso a datos, de tal forma que realmente sean ignorantes de la tecnología subyacente que se utilizará para su persistencia o trabajo. A los objetos entidad que no implementen ninguna clase base y/o interfaz alguna ligadas a la tecnología subyacente se les suele denominar como objetos **POCO (*Plain Old Clr Objects*)** en **.NET**, o **POJO *(Plain Old Java Object*)** en el mundo **Java**. Por el contrario, los objetos de transferencia de datos que sí implementan una determinada clase base o interfaz ligado con la tecnología subyacente, son conocidos por el nombre de 'Clases prescriptivas'. La decisión de decantarse por una alternativa u otra, por supuesto no es algo que uno pueda tomar al azar, más bien todo lo

contrario, debe pensarse detenidamente. Por un lado los objetos POCO nos dan un amplio grado de libertad con respecto al modelo de persistencia que tomemos, de hecho, nada tiene que saber de él, y nos permite intercambiar la información de una forma mucho más transparente, puesto que solamente, en aplicaciones distribuidas, intercambiaríamos un esquema de tipos primitivos, sin conocimiento alguno de una clase de trabajo especial. Como todo no van a ser ventajas, el uso de POCO también lleva restricciones y/o sobrecargas (tradicionalmente suponía un mayor trabajo de desarrollo) asociadas al 'grado de ignorancia' que el motor de persistencia de turno tendrá sobre estas entidades y su correspondencia con el modelo relacional. Las clases POCO suelen tener un mayor coste inicial de implementación, a no ser que el ORM que estemos utilizando nos ayude en cierta generación de clases entidad POCO a partir de un Modelo de Datos del Dominio (Como si hace el ORM de Microsoft, *Entity* Framework 4.0).

El concepto de **IPOCO** (*Interface POCO*) es muy similar al de POCO pero algo más laxo, es decir, las clases de datos que definen las entidades no son completamente 'limpias' sino que dependen de **implementar uno o más interfaces** que especifican qué implementación mínima deben de proporcionar. En este caso (IPOCO) y para cumplir el principio **PI (*Persistance Ignorance*)**, es importante que dicho interfaz esté bajo nuestro control (código propio) y no forme parte de tecnologías externas de Infraestructura. De lo contrario, nuestras entidades dejarían de ser 'agnósticas' con respecto a las capas de Infraestructura y tecnologías externas y pasarían a ser 'Clases Prescriptivas'.

En cualquier caso, las ENTIDADES son objetos flotantes a lo largo de toda la arquitectura o parte de la arquitectura. Pues si hacemos posteriormente uso de DTOs (*Data Transfer Objects*) para las comunicaciones remotas entre *Tiers*, en ese caso, las entidades internas del modelo de dominio no fluirían hasta la capa de presentación ni cualquier otro punto externo a las capas internas del Servicio, serían los objetos DTO los que serían proporcionados a la capa de presentación situada en un punto remoto. El análisis de los DTOs versus entidades, lo realizamos en el capítulo de Servicios Distribuidos, pues son conceptos relacionados con desarrollo distribuido y aplicaciones *N-Tier*.

Por último, considerar requerimientos de serialización de clases que puedan existir de cara a comunicaciones remotas. El pasar entidades de una capa a otra (p.e. de la capa de Servicios Remotos a la Capa de Presentación), requerirá que dichas entidades puedan serializarse, tendrán que soportar algún mecanismo de serialización a formatos tipo XML o binario. Para esto es importante confirmar que el tipo de entidades elegido soporte afectivamente una serialización. Otra opción es, como decíamos, la conversión y/o agregación a DTOs (*Data Transfer Objects*) en la capa de Servicios-Distribuidos.

Lógica interna de la entidad contenida en la propia Entidad

Es fundamental que los propios objetos de ENTIDAD posean también cierta lógica relativa a los datos en memoria de dicha entidad. Por ejemplo, podemos tener lógica de negocio en una entidad de 'CuentaBancaria' donde se realice la suma de dinero cuando

se hace un abono pero también se realicen comprobaciones de la cuenta o de la cantidad a abonar que lógicamente tiene que ser mayor que cero, etc. O lógica de campos calculados y en definitiva, cierta lógica relativa a la parte interna de dicha entidad.

Es posible que algunas clases de nuestras entidades no dispongan de lógica propia, si realmente no lo necesitan. Pero si todas nuestras entidades carecieran completamente de lógica, estaríamos cayendo en el anti-patron '*Anemic Domain Model*' mencionado por Martin Fowler. Ver '***AnemicDomainModel***' de Martin F.:

http://www.martinfowler.com/bliki/AnemicDomainModel.html

El anti-patron '*Anemic-Domain-Model*' se produce cuando solo se tienen entidades de datos como clases que poseen solamente campos y propiedades y la lógica de dominio perteneciente a dichas entidades **está mezclada en clases de nivel superior (Servicios del Dominio o incluso peor, Servicios de Aplicación).** Es importante resaltar que normalmente los Servicios si deben poseer lógica relativa a ENTIDADES pero lógica que trata a dichas entidades como un todo, una unidad o incluso colecciones de dichas unidades. Pero cada ENTIDAD debería poseer la lógica relativa a su 'parte interna', lógica relacionada con sus datos internos en memoria.

Si los SERVICIOS poseyeran absolutamente el 100% de la lógica de las ENTIDADES, esta mezcla de lógica de dominio perteneciente a diferentes entidades sería lo peligroso. Eso sería una implementación 'Transaction Script', completamente contraria al 'Domain Model' u orientación al dominio.

La lógica relativa a consumir/invocar Repositorios de la capa de infraestructura, es lógica que debe de estar normalmente en los SERVICIOS de Aplicación, no del Dominio. Un objeto (ENTIDAD) no tiene qué saber cómo guardarse/construirse a sí mismo, al igual que un motor en la vida real proporciona capacidad de motor, no de fabricarse a sí mismo, o un libro no 'sabe' como guardarse a sí mismo en una estantería.

Tabla 2.- Guía de Arquitectura Marco

Regla Nº: D8.	**Identificación de ENTIDADES basada en la identidad**

Norma

- Cuando a un objeto se le distingue por su identidad y no por sus atributos, dicho objeto debe ser primario en la definición del modelo del Dominio. Debe ser una ENTIDAD. Se debe mantener una definición de clase sencilla y focalizada en la continuidad del ciclo de vida e identidad. Debe tener

alguna forma de distinción aun cuando cambie de atributos o incluso de forma o historia. Relacionado con esta ENTIDAD, deberá existir una operación que garantice el obtener un resultado único para cada objeto, posiblemente seleccionando un identificador único. El modelo debe definir qué significa que sea el mismo objeto ENTIDAD.

Referencias

- ***'ENTITY pattern'* en el libro '*Domain Driven Design*' de Eric Evans.**
- **The Entity Design Pattern**
- http://www.codeproject.com/KB/architecture/entitydesignpattern.aspx

Tabla 3.- Guía de Arquitectura Marco

Regla Nº: D9.	Las ENTIDADES deben ser POCO o IPOCO (En una Arquitectura Domain Oriented o DDD)

Norma

- Para poder cumplir el principio **PI (Persistance Ignorance)** y no tener dependencias directas con tecnologías de infraestructura, es importante que nuestras entidades sean POCO o IPOCO.

Cuando hacer uso de IPOCO

→ Algunas tecnologías ORM permiten hacer uso de POCO e IPOCO, si bien, cuando las clases entidad que nos puedan ser generadas son IPOCO, normalmente nos van a permitir realizar aspectos avanzados (como *Self-Tracking Entities* muy útiles para escenarios N-Tier y gestión de Concurrencia Optimista). Así pues, **para escenarios de aplicaciones N-Tier**, es bastante recomendable el uso de IPOCO por ofrecernos una gran potencia y menos trabajo manual a implementar por nosotros.

Cuando hacer uso de POCO

→ En escenarios puramente SOA, donde la interoperabilidad es crítica, o incluso si queremos que nuestras capas de presentación puedan desarrollarse/cambiarse a un ritmo diferente al Dominio y que cambios en las entidades del Dominio afecten menos a las capas de presentación, es mejor hacer uso de DTOs específicamente creados para los servicios distribuidos y consumidos en las capas de presentación. Si se hace uso de DTOs, lógicamente, los aspectos avanzados de las *Self Tracking Entities* no tienen sentido, así pues, ahí se recomienda el hacer uso de entidades del dominio que sean POCO, que nos ofrece una completa independencia de la capa de persistencia (cumpliendo el principio PI). El uso de DTOs es una orientación al Dominio incluso más pura (gracias al desacoplamiento entre entidades del Dominio y los DTOs que en definitiva serán las entidades de las capas de presentación), pero conlleva un coste y complejidad del desarrollo bastante mayor debido a las conversiones de datos en ambos sentidos desde entidades del dominio a DTOs y viceversa. El uso de entidades IPOCO y Self-Tracking consumiéndose directamente en las capas de presentación es un enfoque más productivo, pero también acopla más al Dominio con las Capas de presentación. Esta decisión (Entidades Self-Tracking vs. DTOs) es una decisión de diseño/arquitectura que dependerá mucho de la magnitud de la aplicación. Si hay varios equipos de desarrollo trabajando para la misma aplicación, probablemente el desacoplamiento de los DTOs será beneficioso.

- Otra última opción es algo mixto. Es decir, hacer uso de Entidades IPOCO/Self-Tacking para aplicaciones N-Tier (comunicación desde capa de presentación, etc.) y simultáneamente disponer de una capa SOA especialmente diseñada para integraciones externas e interoperabilidad, siendo dicha capa SOA ofrecida por lo tanto a otras aplicaciones/servicios externos que consumirían unos servicios-web de integración más simplificados y con DTOs.

Referencias

- ***'ENTITY pattern'*** **en el libro '*Domain Driven Design*' de Eric Evans.**
- **The Entity Design Pattern**

→ http://www.codeproject.com/KB/architecture/entitydesignpattern.aspx

2.2.2.- Patrón Objeto-Valor (‘Value-Object pattern’)

“Muchos objetos no poseen identidad conceptual. Esos objetos describen ciertas características de una cosa”.

Como hemos visto anteriormente, el seguimiento de la identidad de las entidades es algo fundamental, sin embargo, hay muchos otros objetos y datos en un sistema que no necesitan dicha posesión de identidad y tampoco un seguimiento sobre ello. De hecho, en muchos casos no se debería realizar porque puede perjudicar el rendimiento global del sistema en un aspecto, en muchos casos, innecesario. El diseño de software es una constante lucha con la complejidad, y a ser posible, siempre debe minimizarse dicha complejidad. Por lo tanto, debemos hacer distinciones de forma que las gestiones especiales se apliquen solo cuando realmente se necesitan.

La definición de los OBJETO-VALOR es: *Objetos que describen cosas*. Y siendo más precisos, *un objeto sin ninguna identidad conceptual, que describe un aspecto del dominio.* En definitiva, son objetos que instanciamos para representar elementos del diseño y que nos importan solo de forma temporal. Nos importa lo *que* son, no *quienes* son. Ejemplos básicos son los números, los *strings*, etc. Pero también conceptos de más alto nivel. Por ejemplo, una ‘dirección’ en un sistema podría ser una ENTIDAD porque en dicho sistema la dirección es importante como identidad. Pero en otro sistema diferente, la ‘dirección’ puede tratarse simplemente de un OBJETO-VALOR, un atributo descriptivo de una empresa o persona.

Un OBJETO-VALOR puede ser también un conjunto de otros valores o incluso de referencias a otras entidades. Por ejemplo, en una aplicación donde se genere una Ruta para ir de un punto a otro, dicha ruta sería un OBJETO-VALOR (porque sería una ‘foto’ de puntos a pasar por dicha ruta, pero dicha ruta no tendrá identidad ni queremos persistirla, etc.), aun cuando internamente está referenciando a Entidades (Ciudades, Carreteras, etc.).

A nivel de implementación, los OBJETO-VALOR normalmente se pasaran y/o devolverán como parámetros en mensajes entre objetos. Y como decíamos antes, normalmente tendrán una vida corta sin un seguimiento de su identidad.

Asimismo, una entidad suele estar compuesta por diferentes atributos. Por ejemplo, una persona puede ser modelada como una Entidad, con una identidad, e internamente estar compuesta por un conjunto de atributos como el nombre, apellidos, dirección, etc., los cuales son simplemente *Valores*. De dichos valores, los que nos importen como un conjunto (como la dirección), deberemos tratarlos como OBJETO-VALOR.

El siguiente ejemplo muestra un diagrama de clases de una aplicación concreta donde remarcamos qué podría ser una ENTIDAD y qué podría ser posteriormente un OBJETO-VALOR dentro de una ENTIDAD:

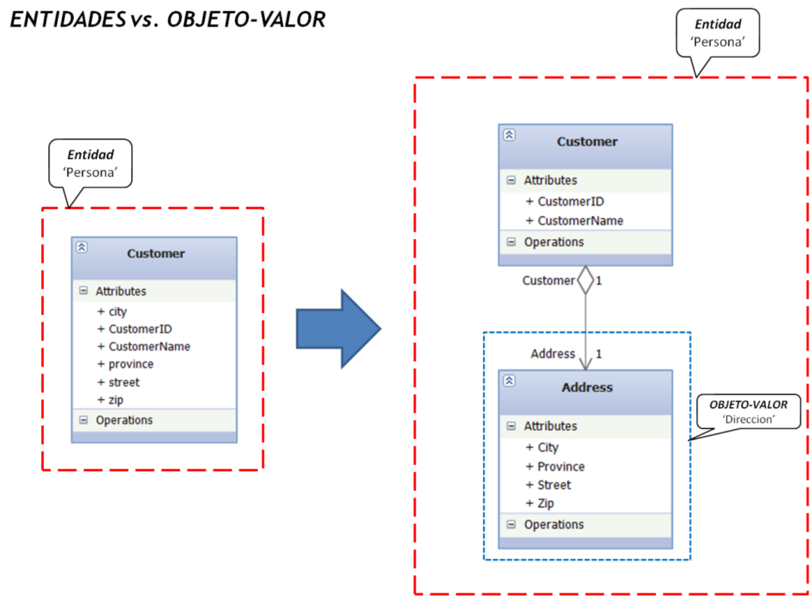


Figura 2.- Entidades vs. Objeto-Valor

Tabla 4.- Guía de Arquitectura Marco

Regla Nº: D10.

Identificar e Implementar el patrón OBJETO-VALOR (VALUE-OBJECT) en los casos necesarios

○ **Recomendaciones**

- Cuando ciertos atributos de un elemento del modelo nos importan de forma agrupada, pero dicho objeto debe carecer de identidad trazable, debemos clasificarlos como OBJETO-VALOR. Hay que expresar el significado de dichos atributos y dotarlos de una funcionalidad relacionada. Así mismo, debemos tratar los OBJETO-VALOR como información inmutable durante toda su vida, desde el momento en el que se crean hasta en el que se destruyen.

Referencias

- **Patrón *'VALUE-OBJECT'.* Por Martin Fowler.** Libro 'Patterns of Enterprise Application Architecture*': "A small simple object, like money or a date range whose equality isn't based on identity"*

- **Patrón *'VALUE-OBJECT'.* Libro *'Domain Driven Design'* - Eric Evans.**

Los atributos que conforman un OBJETO-VALOR deben formar un 'todo conceptual'. Por ejemplo, la calle, ciudad y código postal no deberían ser normalmente simples atributos separados dentro de un objeto persona (Depende del Dominio de la aplicación, por supuesto). Realmente son también parte de una dirección, lo cual simplifica el objeto de la persona y hace más coherente el OBJETO-VALOR.

Sin embargo, este ejemplo puede ser válido dependiendo del caso, en otra aplicación diferente, la dirección podría querer tratarse como ENTIDAD por ser lo suficientemente importante en dicho Dominio como para poseer identidad y trazabilidad de dicha identidad (p.e. un dominio de negocio de una aplicación de compañía eléctrica o telefónica, etc.).

Diseño de OBJETOS VALOR

Debido a la falta de restricciones que tienen los OBJETOS-VALOR, podemos diseñarlos de diferentes formas, siempre favoreciendo a la forma que más simplifique el diseño o que más optimice el rendimiento del sistema. Una de las restricciones de los OBJETO-VALOR debería ser que sus valores deben ser inmutables desde su creación. Por lo tanto, en su creación (constructor) es cuando se le deben proporcionar sus valores y no permitir que se cambien durante la vida del objeto.

Relativo al rendimiento, los OBJETOS-VALOR nos permiten realizar ciertos 'trucos' gracias a su naturaleza de inmutabilidad. Especialmente, en sistemas donde pueden existir miles de instancias de OBJETOS-VALOR con muchas coincidencias de los mismos valores, dicha inmutabilidad nos permitiría reutilizarlos, serían objetos 'intercambiables', porque sus valores son los mismos y no tienen Identidad (como si les pasa a las ENTIDADES). Este tipo de optimizaciones puede a veces marcar la diferencia entre un software lento y otro con un buen rendimiento. Por supuesto, todo esto depende del tipo de entorno y contexto de la aplicación. El compartir objetos a veces puede tener un mejor rendimiento pero en cierto contexto (una aplicación distribuida, por ejemplo) puede ser menos escalable que el disponer de copias, pues el acceder a un punto central de objetos compartidos reutilizables puede suponer un cuello de botella en las comunicaciones.

2.2.3.- Agregados (Patrón 'Aggregate')

Un agregado es un patrón de dominio que se utiliza para definir pertenencia y fronteras de objetos del modelo de dominio.

Un modelo puede tener un número indefinido de objetos (entidades y objetos-valor), y normalmente estarán relacionados entre ellos, incluso de formas complejas, es decir, un mismo objeto entidad puede estar relacionado con varias entidades, no solo con otra entidad. Tendremos, por lo tanto diferentes tipos de asociaciones. Las asociaciones/relaciones entre objetos, se reflejarán en el código e incluso en la base de datos. Por ejemplo, una asociación uno a uno entre un empleado y una compañía, se reflejará como una referencia entre dos objetos e implicará probablemente también una relación entre dos tablas de base de datos. Si hablamos de relaciones uno a muchos, el contexto se complica mucho más. Pero pueden existir muchas relaciones que no sean esenciales para el Dominio concreto en el que estemos trabajando. En definitiva, es difícil garantizar la consistencia en los cambios de un modelo que tenga muchas asociaciones complejas.

Así pues, **uno de los objetivos que debemos tener presente es poder simplificar al máximo el número de relaciones presentes en el modelo de entidades del dominio**. Para esto aparece el concepto o **patrón AGGREGATE**. Un agregado es un grupo/conjunto de objetos asociados que se consideran como una única unidad en lo relativo a cambios de datos. El agregado se delimita por una frontera que separa los objetos internos de los objetos externos. Cada agregado tendrá un objeto raíz que será la entidad raíz y será el único objeto accesible, de forma inicial, desde el exterior. El objeto entidad raíz tendrá referencias a cualquiera de los objetos que componen el agregado, pero un objeto externo solo puede tener referencias al objeto-entidad raíz. Si dentro de la frontera del agregado hay otras entidades (también podrían ser 'objetos-valor'), la identidad de esos objetos-entidad es solo local y tienen solamente sentido perteneciendo a dicho agregado y no de forma aislada.

Precisamente ese único punto de entrada al agregado (entidad raíz) es lo que asegura la integridad de datos. Desde el exterior del agregado no se podrá acceder ni cambiar datos de los objetos secundarios del agregado, solamente a través de la raíz, lo cual implica un nivel de control muy importante. Si la entidad raíz se borra, el resto de objetos del agregado debería borrarse también.

Si los objetos de un agregado deben poder persistirse en base de datos, entonces solo deben poder consultarse a través de la entidad raíz. Los objetos secundarios deberán obtenerse mediante asociaciones transversales. Esto implica que solo las entidades raíz de un agregado (o entidades sueltas), podrán tener REPOSITORIOS asociados. Lo mismo pasa en un nivel superior con los SERVICIOS. Podremos tener SERVICIOS directamente relacionados con la entidad raíz de un AGREGADO, pero nunca directamente con solo un objeto secundario de un agregado.

Lo que si se debe permitir es que los objetos internos de un agregado tengan referencias a entidades raíz de otros agregados (o a entidades simples).

A continuación mostramos un ejemplo de agregado en el siguiente diagrama:

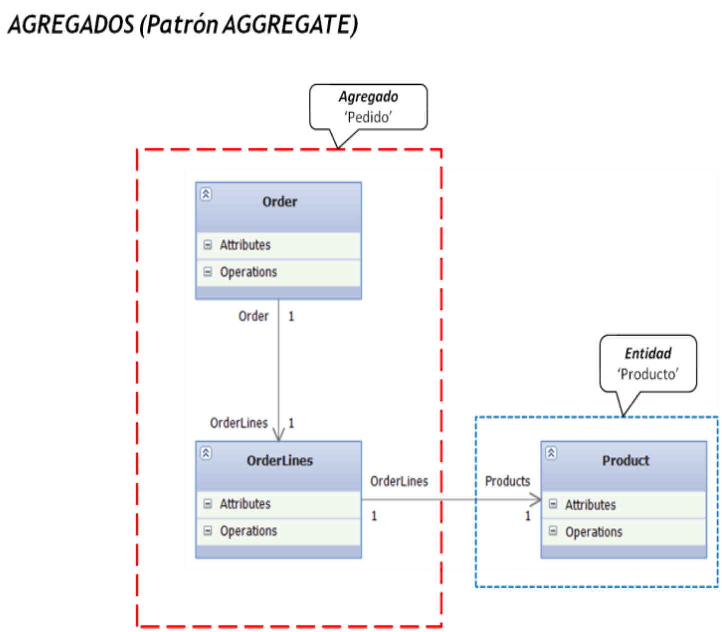


Figura 3.- AGREGADOS (Patrón AGGREGATE)

Tabla 5.- Regla de identificación de Agregados

Regla Nº: D11.	**Identificar e Implementar el patrón AGREGADO (AGGREGATE) en los casos necesarios para simplificar al máximo las relaciones entre objetos del modelo**

○ **Recomendaciones**

- **Uno de los objetivos que debemos tener presente es poder simplificar al máximo el número de relaciones presentes en el modelo de entidades del dominio. Para esto aparece el concepto o patrón AGGREGATE.** Un agregado es un grupo/conjunto de objetos asociados que se consideran como una única unidad en lo relativo a cambios de datos.

- Tener muy presente que esto implica que **solo las entidades raíz de un agregado (o también las entidades simples), podrán tener REPOSITORIOS asociados. Lo mismo pasa en un nivel superior con los SERVICIOS. Podremos tener SERVICIOS directamente relacionados con la entidad raíz de un AGREGADO, pero nunca directamente con solo un objeto secundario de un agregado.**

Referencias

- **Patrón '*AGGREGATE*'.** Libro *'Domain Driven Design'* - Eric Evans.

2.2.4.- Contratos/Interfaces de Repositorios dentro de la Capa de Dominio

La implementación de los Repositorios no es parte del Dominio sino parte de las capas de Infraestructura (puesto que los Repositorios están ligados a una tecnología de persistencia de datos, como ORMs tipo *Entity Framework*), sin embargo, el 'contrato' de como deben ser dichos Repositorios (Interfaces a cumplir por dichos Repositorios), eso si debe formar parte del Dominio. Por eso lo incluimos aquí. Esto es así porque dicho contrato especifica qué debe ofrecer el Repositorio, sin importarme como está implementado por dentro. Dichos interfaces sí son agnosticos a la tecnología. Así pues, los interfaces de los Repositorios es importante que estén definidos dentro de las Capas del Dominio. Este punto es algo precísamente recomendado en las arquitecturas DDD y está basado en el patrón '*Separated Interface Pattern*' definido por Martin Fowler.

Lógicamente, para poder cumplir este punto, es necesario que las 'Entidades del Dominio' y los 'Value-Objects' sean **POCO/IPOCO**, es decir, también completamente agnosticos a la tecnología de acceso a datos. Hay que tener en cuenta que las entidades del dominio son, al final, los 'tipos de datos' de los parámetros enviados y devueltos por y hacia los Repositorios.

En definitiva, con este diseño (*Persistence Ignorance*) lo que buscamos es que las clases del dominio 'no sepan nada directamente' de los repositorios. Cuando se trabaja en las capas del dominio, se debe ignorar como están implementados los repositorios.

Tabla 6.- Guía de Arquitectura Marco

Regla Nº: D12.	**Definir interfaces de Repositorios dentro de la Capa de Dominio siguiendo el patrón INTERFAZ-SEPARADO (SEPARATED-INTERFACE PATTERN)**

○ **Recomendaciones**

- Desde el punto de vista de desacoplamiento entre la capa de Dominio y la de Infraestructura de acceso a Datos, se recomienda definir los interfaces de los

- Repositorios dentro de la Capa de dominio, y la implementación de dichos dominios dentro de la Capa de Infraestructura de Persistencia de Datos. De esta forma, una clase del Modelo de Dominio podrá hacer uso de un interfaz de Repositorio que necesite, sin tener que conocer la implementación de Repositorio actual, que estará implementado en la capa de Infraestructura.

- Esta regla, encaja perfectamente con las técnicas de desacoplamiento basadas en contenedores IoC.

Referencias

- **Patrón '*Separated Interface*'. Por Martin Fowler.**
  "Use Separated Interface to define an interface in one package but implement it in another. This way a client that needs the dependency to the interface can be completely unaware of the implementation."
  http://www.martinfowler.com/eaaCatalog/separatedInterface.html

2.2.5.-SERVICIOS del Modelo de Dominio

En la mayoría de los casos, nuestros diseños incluyen operaciones que no pertenecen conceptualmente a objetos de ENTIDADES del Dominio. En estos casos podemos incluir/agrupar dichas operaciones en SERVICIOS explícitos del Modelo del Dominio.

Nota:
Es importante destacar que el concepto SERVICIO en capas N-Layer DDD no es el de SERVICIO-DISTRIBUIDO (Servicios Web normalmente) para accesos remotos. Es posible que un Servicio-Web 'envuelva' y publique para accesos remotos a la implementación de Servicios del Dominio, pero también es posible que una aplicación web disponga de servicios del dominio y no disponga de Servicio Web alguno.

Dichas operaciones que no pertenecen específicamente a ENTIDADES del Dominio, son intrínsecamente actividades u operaciones, no características internas de entidades del Dominio. Pero debido a que nuestro modelo de programación es orientado a objetos, debemos agruparlos también en objetos. A estos objetos les llamamos SERVICIOS.

El forzar a dichas operaciones del Dominio (en muchos casos son operaciones de alto nivel y agrupadoras de otras acciones) a formar parte de objetos ENTIDAD distorsionaría la definición del modelo del dominio y haría aparecer ENTIDADES artificiales.

Un SERVICIO es una operación o conjunto de operaciones ofrecidas como un interfaz que simplemente está disponible en el modelo.

La palabra "Servicio" del patrón SERVICIO precisamente hace hincapié en lo que ofrece: "Qué puede hacer y qué acciones ofrece al cliente que lo consuma y enfatiza la relación con otros objetos del Dominio (Englobando varias Entidades, en muchos casos)".

A los SERVICIOS de alto nivel (relacionados con varias entidades) se les suele nombrar con nombres de Actividades. En esos casos, están por lo tanto relacionados con verbos de los Casos de Uso del análisis, no con sustantivos, aun cuando puede tener una definición abstracta de una operación de negocio del Dominio (Por ejemplo, un Servicio-Transferencia relacionado con la acción/verbo 'Transferir Dinero de una cuenta bancaria a otra').

Los nombres de las operaciones de un SERVICIO deben surgir del LENGUAJE UBICUO del Dominio. Los parámetros y resultados obtenidos deben ser objetos del Dominio (ENTIDADES u OBJETOS-VALOR).

Las clases SERVICIO son también componentes del dominio, pero en este caso de un nivel superior, en muchos casos abarcando diferentes conceptos y ENTIDADES relacionadas con escenarios y casos de uso completos.

Cuando una operación del Dominio se reconoce como concepto importante del Dominio, normalmente deberá incluirse en un SERVICIO del Dominio.

Los servicios no deben tener estados. Esto no significa que la clase que lo implementa tenga que ser estática, podrá ser perfectamente una clase instanciable (y necesitaremos que NO sea estática si queremos hacer uso de técnicas de desacoplamiento entre capas, como contenedores IoC). Que un SERVICIO sea *stateless* significa que un programa cliente puede hacer uso de cualquier instancia de un servicio sin importar su historia individual como objeto.

Adicionalmente, la ejecución de un SERVICIO hará uso de información que es accesible globalmente y puede incluso cambiar dicha información global (es decir, podría tener efectos colaterales). Pero el servicio no contiene estados que pueda afectar a su propio comportamiento, como si tienen la mayoría de los objetos del dominio.

En cuanto al tipo de reglas a incluir en los SERVICIOS del Dominio, un ejemplo claro sería en una aplicación bancaria, el realizar una transferencia de una cuenta a otra, porque requiere de una coordinación de reglas de negocio relativas a ENTIDADES de tipo 'Cuenta' y operaciones a coordinar de tipo 'Abono' y 'Cargo'. Además, la acción/verbo 'Transferir' es una operación típica del Dominio bancario. En este caso, el SERVICIO en si no realiza mucho trabajo, simplemente coordinará las llamadas a los métodos Cargar() y Abonar() probablemente de las clases ENTIDAD de más bajo nivel como 'CuentaBancaria'. Pero en cambio, el situar el método Transferir() o RealizarTransferencia() en un objeto 'Cuenta' sería en principio erróneo (por supuesto, esto depende del Dominio concreto) porque la operación involucra a dos 'Cuentas' y posiblemente a otras reglas de negocio a tener en cuenta.

Tanto en los Servicios del Dominio como en la lógica interna de las clases entidad deberá implementarse la <u>generación y gestión de excepciones de negocio.</u>

Desde el punto de vista externo al dominio, serán normalmente los SERVICIOS los que deben ser visibles para realizar las tareas/operaciones relevantes de cada capa, en el ejemplo anterior (Transferencia Bancaria), el SERVICIO es precisamente la columna vertebral de las reglas de negocio del Dominio bancario de nuestro ejemplo.

Tabla 7.- Guía de Arquitectura Marco

Regla Nº: D13.	**Diseñar e implementar SERVICIOS del Dominio para coordinar la lógica de negocio**

<u>Recomendaciones</u>

- Es importante que existan estos componentes para poder coordinar la lógica del dominio de las entidades, así como para no mezclar nunca la lógica del dominio (reglas de negocio) con la lógica de aplicación y acceso a datos (persistencia ligada a una tecnología).

- Un buen SERVICIO suele poseer estas tres características:

    - La operación está relacionada con un concepto del Dominio que no es una parte natural de la lógica <u>interna</u> relacionada con una ENTIDAD u OBJETO VALOR

    - El interfaz de acceso está definido basado en varios elementos del modelo de dominio.

Referencias

- **SERVICE Pattern** - Libro *'Domain Driven Design'* - **Eric Evans**.

- **SERVICE LAYER Pattern** – Por **Martin Fowler**. Libro '*Patterns of Enterprise Application Architecture*': "*Layer of services that establishes a set of available operations and coordinate the application's response in each main operation*"

Otra regla a tener en cuenta de cara a la definición de entidades de datos e incluso de clases y métodos, es ir definiendo lo que realmente vamos a utilizar, no definir entidades y métodos porque nos parece lógico, porque probablemente al final mucho de eso no se utilizará en la aplicación. En definitiva es seguir la recomendación en metodologías ágiles denominada "**YAGNI**" (*You ain't gonna need it*), mencionada al principio de esta guía.

Debemos definir Servicios del Dominio solamente ahí donde lo necesitemos, donde aparezcan necesidades de coordinación de lógica de dominio de las entidades.

Como se puede observar en el gráfico siguiente, podemos tener un servicio del dominio (en este caso la clase de servicio *BankTransferService*) que coordine acciones de lógica de negocio de las cuentas bancarias (Clase entidad *BankAccount*):

Relación entre objetos de Servicios y Entidades del Dominio

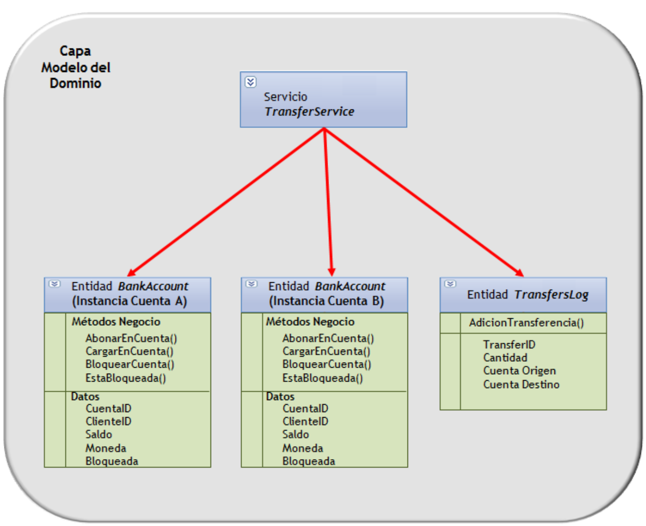


Figura 4.- Posible relación entre objetos de Servicio y Entidades

En UML, con un diagrama de secuencia simplificado (sin tener en cuenta un registro de transferencias), tendríamos la siguiente interacción. Básicamente solo

destacar que las llamadas entre métodos en esta capa serán exclusivamente para efectuar lógica del Dominio cuyo flujo o interacción podríamos discutir con un experto del Dominio o usuario final:

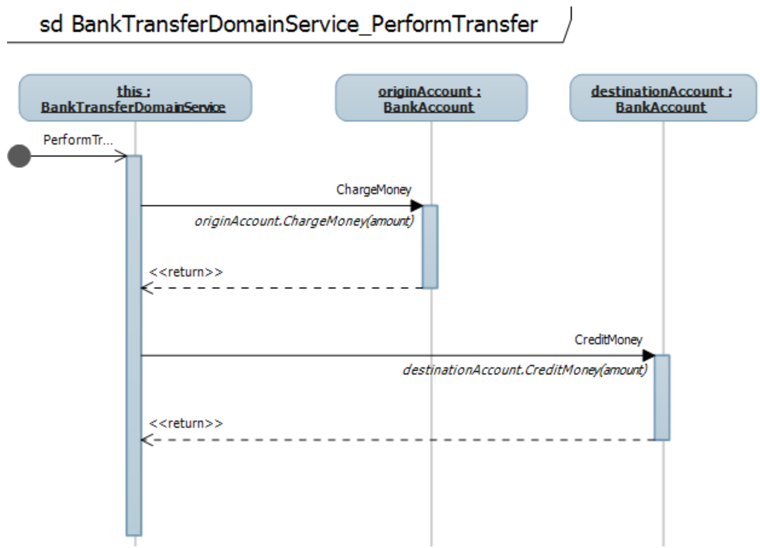


Figura 5.- Diagrama de secuencia

En los tres objetos que aparecen en el diagrama de secuencia, el primero (*BankTransferDomainService*) y origen de la secuencia es un servicio del Dominio y los otros dos (originAccount y destinationAccount, ambos de la clase BankAccount) son objetos 'Entidad de Dominio', los cuales dispondrían de métodos/lógica del dominio (los métodos ChargeMoney() y CreditMoney()) que modificarán los datos en memoria que tiene cada objeto-entidad del dominio.

Tabla 8.- Servicios del Dominio deben regir/coordinar la lógica de negocio

Regla Nº: D14.	**Las clases SERVICIO del Dominio deben también regir/coordinar los procesos principales del Dominio**

○ **Norma**

- Como norma general, todas las operaciones de negocio complejas (**que requieran más de una operación unitaria**) relativas a diferentes Entidades del Dominio, deberán implementarse en clases SERVICIO del

Dominio.

- En definitiva, se trata de implementar la lógica de negocio de los escenarios y casos de uso completos.

Referencias

SERVICE Pattern - '*Domain Driven Design*' - Eric Evans.
SERVICE LAYER Pattern – Por Martin Fowler. Libro '*Patterns of Enterprise Application Architecture*': "*Layer of services that establishes a set of available operations and coordinate the application's response in each main operation*"

Tabla 9.- Implementar solo coordinación de lógica del Dominio

Regla Nº: D15.

Implementar solo coordinación de lógica del Dominio en los Servicios del Dominio

○ **Recomendación**

- Es fundamental que la lógica de los Servicios del Dominio sea código muy limpio, simplemente las llamadas a los componentes de más bajo nivel (Lógica de clases entidad, normalmente), es decir, simplemente las acciones que explicaríamos o nos confirmaría un experto en el Dominio/Negocio. Normalmente (salvo excepciones) no se debe implementar aquí ningún tipo de coordinación de acciones de aplicación/infraestructura, como llamadas a Repositorios, creación de transacciones, uso de objeto UoW, etc. Estas otras acciones de coordinación de la 'fontanería' de nuestra aplicación debe implementarse en los Servicios de la Capa de Aplicación.

- Esto es una recomendación para que las clases del Dominio queden mucho más limpias. Pero es perfectamente viable (muchas arquitecturas N-Capas incluso DDD lo realizan así), mezclar código de coordinación de persistencia, UoW y transacciones con código de lógica de negocio de Servicios del Dominio.

- Implementar Servicios del Dominio solo si son necesarios (YAGNI).

2.2.6.-Patrón ESPECIFICACION (SPECIFICATION)

El enfoque del patrón **ESPECIFICACION** consiste en **separar la sentencia de qué tipo de objetos deben ser seleccionados en una consulta del propio objeto que realiza la selección**. El objeto 'Especificación' tendrá una responsabilidad clara y limitada que deberá estar separada y desacoplada del objeto de Dominio que lo usa.

Este patrón está explicado a nivel lógico y en detalle, en un '*paper*' conjunto de Martin Fowler y Eric Evans: **http://martinfowler.com/apsupp/spec.pdf**

Así pues, la idea principal es que la sentencia de 'que' datos candidatos debemos obtener debe de estar separada de los propios objetos candidatos que se buscan y del mecanismo que se utilice para obtenerlos.

Vamos a explicar este patrón a continuación de forma lógica según originalmente fue definido por Martin y Eric, **sin embargo, en el sección de 'Implementación de Capa de Dominio' veremos que la implementación elegida por nosotros difiere en parte del patrón lógico original debido a la mayor potencia de lenguaje que se nos ofrece en .NET y en concreto con los árboles de expresiones, donde podemos obtener un mayor beneficio que si trabajamos solamente con especificaciones para objetos en memoria, como lo describen MF y EE**. No obstante, nos parece interesante explicarlo aquí según su definición original para comprender bien la esencia de este patrón.

Casos en los que es muy útil el patrón ESPECIFICACION (SPECIFICATION)

En las aplicaciones en las que se permita a los usuarios realizar consultas abiertas y compuestas y 'grabar' dichos tipos de consultas para disponer de ellas en un futuro (p.e. un analista de clientes que se guarda una consulta compuesta por él que muestre solo los clientes de cierto país que han hecho pedidos mayores de 200.000€, y otro tipo de condiciones que él mismo ha seleccionado, etc.), en esos casos son especialmente útiles las ESPECIFICACIONES.

Patrón Subsunción (Patrón relacionado)

Una vez que usamos el patrón **SPECIFICATION**, otro patrón muy útil es el patrón **SUBSUMPTION**, en Español, **SUBSUNCION**, ciertamente, una palabra muy poco común en nuestro idioma. (**Subsunción**: Acción y efecto de subsumir. De sub- y el latín 'sumĕre', tomar. Incluir algo como componente en una síntesis o clasificación más abarcadora. Considerar algo como parte de un conjunto más amplio o como caso particular sometido a un principio o norma general. -*Diccionario de la Real Academia de la Lengua Española*-).

Es decir, el uso normal de especificaciones prueba la especificación contra un objeto candidato para ver si ese objeto satisface todos los requerimientos expresados en la especificación. La 'Subsunción' permite comparar especificaciones para ver si satisfaciendo una especificación eso implica la satisfacción de otra segunda

especificación. También es posible a veces el hacer uso del patrón 'Subsunción' para implementar la satisfacción. Si un objeto candidato puede producir una especificación que lo caracteriza, el probar con una especificación entonces viene a ser una comparación de especificaciones similares. La 'Subsunción' funciona especialmente bien en Aplicaciones Compuestas (Composite-Apps).

Como este concepto lógico de SUBSUNCION empieza a complicarnos bastante las posibilidades, lo mejor es ver la tabla clarificadora que nos ofrecen *Martin Fowler y Eric Evans* en su '*paper*' público sobre qué patrón utilizar y cómo dependiendo de las necesidades que tengamos:

Tabla 10.- Tabla clarificadora patrón SPECIFICATION – Por Martin Fowler

Problemas	Solución	Patrón
- Necesitamos seleccionar un conjunto de objetos basándonos en un criterio concreto. - Necesitamos comprobar que solo se utilizan ciertos objetos por ciertos roles. - Necesitamos describir que puede hacer un objeto sin explicar los detalles de cómo lo hace el objeto y describiendo la forma en que un candidato podría construirse para satisfacer el requerimiento.	- Crear una especificación que sea capaz de mostrar si un objeto candidato coincide con cierto criterio. La especificación tiene un método Bool IsSatisfiedBy(Objeto): que devuelve un 'true' si todos los criterios han sido satisfechos por el 'Objeto'	ESPECIFICACION (Specification)
- ¿Cómo implementamos una ESPECIFICACION?	- Codificamos los criterios de selección en el método IsSatisfiedBy() como un bloque de código. - Creamos atributos en la especificación para valores que normalmente varien. Codificamos el método IsSatisfiedBy() para combinar	ESPECIFICACION 'Hard Coded' ESPECIFICACION parametrizada

	esos parámetros y pueda realizarse la prueba. - Crear elementos 'hoja' para los diferentes tipos de pruebas. - Crear nodos compuestos para los operadores 'and', 'or', 'not' (Ver Combinación de Especificaciones, más abajo).	ESPECIFICACIONES COMPUESTAS
- ¿Cómo comparar dos especificaciones para ver si una es un caso especial de otra o si es sustituible por otra?	- Crear una operación llamada isGeneralizationOf(Specification) que contestará si el receptor es en todos los sentidos igual o más general que el argumento.	SUBSUNCION
- Necesitamos descubrir qué debe hacerse para satisfacer los requerimientos. - Necesitamos explicar al usuario por qué la Especificación no ha sido satisfecha.	- Añadir un método llamado RemainderUnsatisfiedBy() que devuelva una Especificación que esprese solo los requerimientos que no deben cumplirse por el objeto objetivo. (A usarse mejor con la Especificación Compuesta).	ESPECIFICACION PARCIALMENTE SATISFECHA

Tabla 11.- Cuando hacer uso del Patrón ESPECIFICACION

Regla Nº: D16.

Hacer uso del Patrón ESPECIFICACION en el diseño e implementación de consultas abiertas y/o compuestas

○ **Norma**

- Identificar partes de la aplicación donde este patrón es útil y hacer uso de él en el diseño e implementación de los componentes del Dominio (Especificaciones) e implementación de ejecución de las especificaciones

(Repositorios).

✓ **Cuando hacer uso del Patrón SPECIFICATION**

PROBLEMA

- *Selección*: Necesitamos seleccionar un conjunto de objetos basados en ciertos criterios y 'refrescar' los resultados en la aplicación en ciertos intervalos de tiempo.

- *Validación*: Necesitamos comprobar que solo los objetos apropiados se utilizan para un propósito concreto.

- *Construcción a solicitar*: Necesitamos describir qué podría hacer un objeto sin explicar los detalles de cómo lo hace, pero de una forma que un candidato podría construirse para satisfacer el requerimiento.

SOLUCIÓN

- Crear una especificación que sea capaz de decir si un objeto candidato cumple ciertos criterios. La especificación tendrá un método bool IsSatisfiedBy(unObjeto) que devuelve true si todos los criterios se cumplen por dicho objeto.

Ventajas del uso de 'Especificaciones'

- Desacoplamos el diseño de los requerimientos, cumplimiento y validación.

- Permite una definición de consultas clara y declarativa.

Cuando no usar el patrón 'Especificación'

- Podemos caer en el anti-patrón de sobre utilizar el patrón ESPECIFICACION y utilizarlo demasiado y para todo tipo de objetos. Si nos encontramos que no estamos utilizando los métodos comunes del patrón SPECIFICATION o que nuestro objeto especificación está representando realmente una entidad del dominio en lugar de situar restricciones sobre otros, entonces debemos reconsiderar el uso de este patrón.

- En cualquier caso, no debemos utilizarlo para todo tipo de consultas, solo

para las que identificamos como idóneas para este patrón. No debemos sobre-utilizarlo.

Referencias

- *'Paper'* conjunto de Martin Fowler y Eric Evans: **http://martinfowler.com/apsupp/spec.pdf**

La definición original de este patrón, mostrada en el diagrama UML siguiente, muestra que se trabaja con objetos y/o colecciones de objetos que deben satisfacer una especificación.

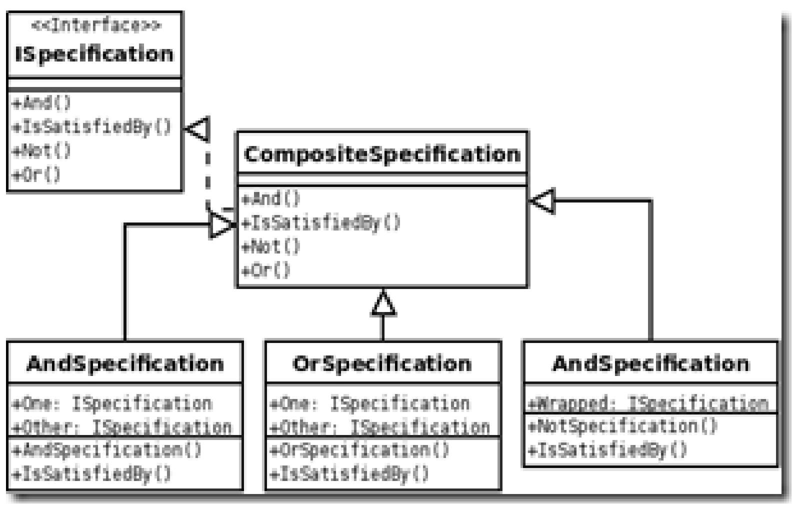


Figura 6.- Diagrama UML del patrón Specification

Esto es precisamente lo que comentábamos que no tiene sentido en una implementación avanzada con .NET y EF (u otro ORM) donde podemos trabajar con consultas que directamente trabajarán contra la base de datos en lugar de objetos en memoria, como pre-supone originalmente el patrón SPECIFICATION.

La razón principal de la afirmación anterior viene de la propia definición del patrón, la cual implica trabajar con objetos directamente en memoria puesto que el método **IsSatisfiedBy()** tomaría una instancia del objeto en el cual queremos comprobar si cumple un determinado criterio y devolver **true** o **false** según se cumpla o no, algo que por supuesto no deseamos por la sobrecarga que esto implicaría. Por todo esto podríamos modificar un poco nuestra definición de ESPECIFICACION para que en

vez de devolver un booleano negando o afirmando el cumplimiento de una especificación determinada, pudiéramos devolver una "*expression*" con el criterio a cumplir.

Este punto lo extendemos más y explicamos en detalle en nuestra implementación del patrón SPECIFICATION en el capítulo de 'Implementación de la Capa de Dominio'.

2.3.- Consideraciones de Diseño de la Capa de Dominio

Al diseñar las sub-capas del Dominio, el objetivo principal de un arquitecto de software debe ser minimizar la complejidad separando las diferentes tareas en diferentes áreas de preocupación/responsabilidad (*concerns*), por ejemplo, los procesos de negocio, entidades, etc., todos ellos representan diferentes áreas de responsabilidad. Dentro de cada área, los componentes que diseñemos deben centrarse en dicha área específica y no incluir código relacionado con otras áreas de responsabilidad.

Se deben considerar las siguientes guías a la hora de diseñar las capas de negocio:

- **Definir diferentes tipos de componentes del Dominio**. Siempre es una buena idea disponer de diferentes tipos de objetos que implementen diferentes tipos de patrones, por tipos de responsabilidad. Esto mejorará el mantenimiento y reutilización de código de la aplicación. Por ejemplo, podemos definir clases de SERVICIOS del dominio, y otros componentes diferenciados para los contratos de ESPECIFICACIONES de consultas o por supuesto, las clases de ENTIDADES del Dominio, también diferenciadas. Finalmente, incluso podremos tener ejecuciones de procesos de negocio de tipo *worklow* (un *workflow* con reglas de negocio dinámicas, etc.), aunque normalmente nos interesará situar los *Workflows* de coordinación a un nivel superior, en la capa de Aplicación, no en esta capa del Dominio.

- **Identificar las responsabilidades de las reglas del dominio**. Se debe de usar la capa del dominio para procesar reglas de negocio, transformar datos por requerimientos de lógica del dominio, aplicar políticas, e implementar validaciones relativas a requerimientos de negocio.

- **Diseñar con alta cohesión**. Los componentes deben contener solo funcionalidad específica (*concerns*) relacionada con dicho componente o entidad.

- **No se deben mezclar diferentes tipos de componentes en las capas del dominio**. Se deben utilizar las capas del dominio para desacoplar la lógica de negocio de la presentación y del código de acceso a datos, así como para simplificar las pruebas unitarias de la lógica de negocio. Esto finalmente aumentará drásticamente la **mantenibilidad** del sistema

- **Reutilizar lógica de negocio común**. Es bueno utilizar estas capas de negocio para centralizar funciones de lógica de negocio reutilizable por diferentes tipos de aplicaciones cliente (Web, RIA, Móvil, etc.).

- **Identificar los consumidores de las capas del Dominio**. Esto ayudará a determinar cómo exponer las capas de negocio. Por ejemplo, si la capa de presentación que va a consumir las capas de negocio fuera una aplicación Web tradicional, probablemente lo más óptimo es acceder directamente. Sin embargo, si la capa de presentación se ejecuta en máquinas remotas (aplicaciones RIA y/o *RichClient*), será necesario exponer las capas del Dominio y Aplicación a través de una capa de Servicios Distribuidos (Servicios Web).

- **Hacer uso de abstracciones para implementar interfaces desacoplados**. Esto se consigue con componentes de tipo 'interfaz', definiciones comunes de interfaces o abstracciones compartidas donde componentes concretos dependen de abstracciones (Interfaces) y no de otros componentes concretos, es decir, no dependen directamente de clases (Esto enlaza con el principio de Inyección de dependencias para realizar desacoplamiento). Sobre todo es importante para los SERVICIOS del Dominio.

- **Evitar dependencias circulares**. Las capas del dominio de negocio solo deben 'conocer' detalles relativos a las capas inferiores (interfaces de Repositorios, etc.) y siempre, a ser posible, a través de abstracciones (interfaces) e incluso mediante contenedores IoC, pero no deben 'conocer' directamente absolutamente nada de las capas superiores (Capa de Aplicación, Capa de Servicios, Capas de Presentación, etc.).

- **Implementar un desacoplamiento entre las capas del dominio y capas inferiores (Repositories) o superiores**. Se debe hacer uso de abstracciónes cuando se cree un interfaz para las capas de negocio. La abstracción se puede implementar mediante interfaces públicos, definiciones comunes de interfaces, clases base abstractas o mensajería (Servicios Web o colas de mensajes). Adicionalmente, las técnicas más potentes para conseguir desacoplamiento entre capas internas son, IoC (Inversion Of Control) y DI (Inyección de Dependencias).

2.4.- EDA y Eventos del Dominio para articular reglas de negocio

Nota:
Es importante destacar que en la versión actual (V1.0) de la implementación de esta Arquitectura de referencia y su aplicación ejemplo V1.0, no hacemos uso de eventos y EDA. Sin embargo, nos parece interesante ir introduciendo el concepto de 'Orientación a Eventos' y Event-Sourcing como otros posibles diseños e implementaciones.

Relacionado con **EDA** (*Event Driven Architecture*), en un dominio de aplicación existirán muchas reglas de negocio de tipo 'condición', por ejemplo, si un cliente ha realizado compras por más de 100.000€, recibir ofertas o trato diferencial, en definitiva, realizar cualquier acción extra. Esto es completamente una regla de negocio, lógica del dominio, pero la cuestión es que podríamos implementarlo de diferentes maneras.

Implementación Condicional (Código Tradicional con sentencias de control)

Podríamos, simplemente, implementar dicha regla mediante una sentencia de control condicional (tipo if..then), sin embargo, este tipo de implementación puede volverse tipo 'espagueti' según vamos añadiendo más y más reglas del dominio. Es más, tampoco tenemos un mecanismo de 'reutilización' de condiciones y reglas a lo largo de diferentes métodos de diferentes clases del dominio.

Implementación Orientada a Eventos del Dominio (Código Condicional Tradicional)

Realmente, para el ejemplo puesto, queremos algo así:

"*Cuando un Cliente es/tiene [algo] el sistema debe [hacer algo]*"

Ese caso es realmente un caso que un modelo basado en eventos lo podría coordinar muy bien. De esa forma, si queremos realizar más cosas/acciones en el "hacer algo" podríamos implementarlo fácilmente como un gestor de eventos adicional…

Los **Eventos del Dominio** sería, en definitiva, algo similar a esto:

"*Cuando un [algo] se ha producido, el sistema debe [hacer algo]*"…

Por supuesto, podríamos implementar dichos eventos en las propias entidades, pero puede ser muy ventajoso disponer de estos eventos a nivel de todo el dominio.

A la hora de implementarlo, debemos implementar un evento a nivel global y en cada regla de negocio implementar una suscripción a dicho evento y eliminar posteriormente la suscripción a dicho evento del dominio.

La mejor forma de implementarlo es teniendo cada evento gestionado por una única clase que no está ligada a ningún caso de uso específico, pero que puede ser activada de forma genérica según lo necesiten los diferentes casos de uso.

2.4.1.-Eventos del Dominio Explícitos

Estos eventos globales del dominio pueden implementarse mediante clases específicas para que cualquier código del dominio pueda lanzar uno de dichos eventos. Esta capacidad la podemos implementar mediante clases estáticas que hagan uso de un contenedor IoC y DI (como Unity, Castle o Spring.NET). Esta implementación la mostramos en el capítulo correspondiente de implementación de capas de lógica del dominio.

2.4.2.-Testing y Pruebas Unitarias cuando utilizamos Eventos del Dominio

El hecho de hacer uso de eventos del dominio, puede complicar y perjudicar las pruebas unitarias de dichas clases del dominio, pues necesitamos hacer uso de un contenedor IoC para comprobar qué eventos del dominio se han lanzado.

Sin embargo, implementando ciertas funcionalidades a las clases de eventos del dominio, podemos solventar este problema y llegar a realizar pruebas unitarias de una forma auto-contenida sin necesidad de un contenedor. Esto se muestra también en el capítulo de implementación de capas de lógica del dominio.

3.- IMPLEMENTACIÓN DE LA CAPA DE DOMINIO CON .NET 4.0

El objetivo del presente capítulo es mostrar las diferentes opciones que tenemos a nivel de tecnología para implementar la 'Capa de Dominio' y por supuesto, explicar la opción tecnológica elegida por defecto en nuestra Arquitectura Marco .NET 4.0, de referencia.

En el siguiente diagrama de Arquitectura resaltamos la situación de la Capa de Dominio en un diagrama *Layer* de Visual Studio 2010:

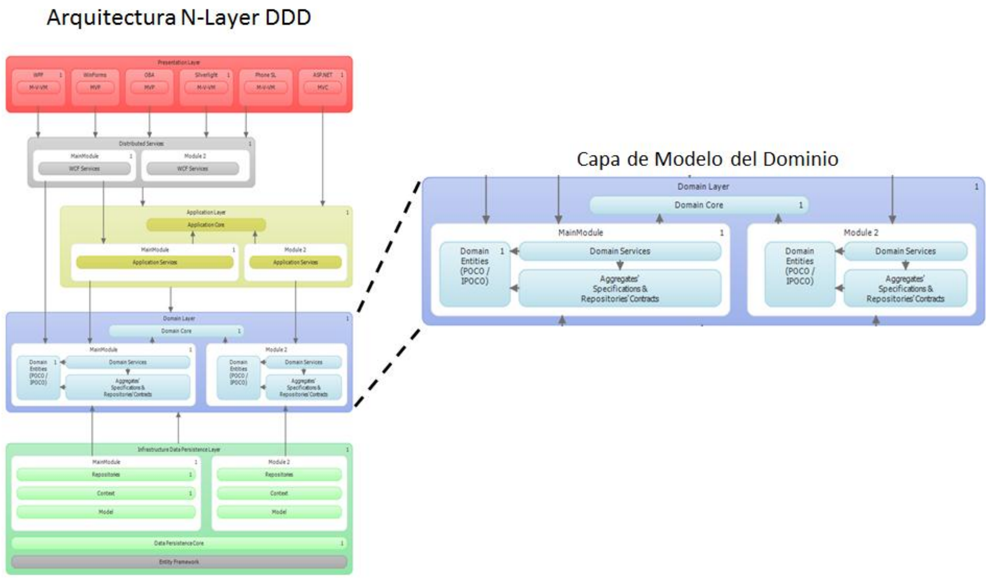


Figura 7.- Situación de Capa de Dominio en Diagrama de Arquitectura VS.2010

3.1.- Implementación de Entidades del Dominio

El primer punto es seleccionar una tecnología para implementar las 'Entidades del Dominio'. Las entidades se usan para contener y gestionar los datos principales de nuestra aplicación. En definitiva, las entidades del dominio son clases que contienen valores y los exponen mediante propiedades, pero también pueden y deben exponer métodos con lógica de negocio de la propia entidad.

En el siguiente sub-esquema resaltamos donde se sitúan las entidades dentro de la Capa de Modelo del Dominio, en el diagrama *Layer* realizado con Visual Studio 2010:

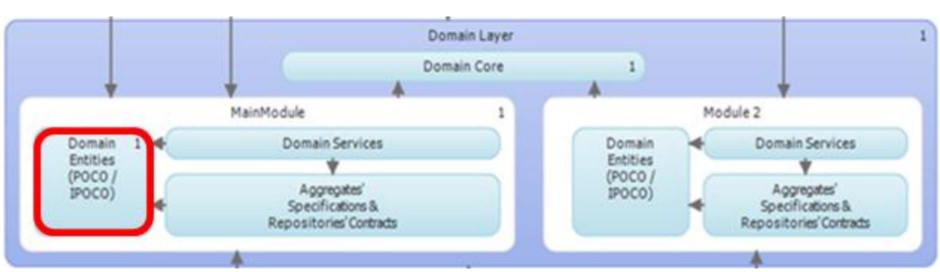


Figura 8.- Sub-Esquema Entidades del Dominio

La elección del tipo de dato/tecnología y formato a utilizar para nuestras entidades del dominio es muy importante pues determina aspectos a los que afectará como las siguientes preguntas:

- ¿Es independiente nuestra Capa de Dominio de la tecnología de acceso a datos? ¿Podríamos dejar de utilizar nuestra actual tecnología de acceso a datos y pasar a una tecnología futura y seguir utilizando nuestras clases .NET de nuestras entidades del dominio? La contestación a esto es muy diferente si estamos utilizando como entidades del dominio Datasets, clases custom, clases prescriptivas de EF o clases POCO/IPOCO.

- ¿Podríamos mantener las mismas entidades del dominio y cambiar a una tecnología diferente/nueva? Por ejemplo, haciendo uso de las mismas entidades podría pasar de LinqToSQL a Entity-Framework, o de NHibernate a Entitiy-Framework, pero si usamos DataSets como entidades, entonces seguro que no podremos cambiar simplemente nuestro sistema de acceso a datos y requeriremos de una reestructuración completa de nuestra aplicación, lo que afectará de lleno al corazón de nuestra aplicación: La Capa de Modelo del Dominio.

- En el caso de no hacer uso de DTOs sino que las entidades del dominio viajen también a la capa de presentación, la elección de las entidades es también crítica de cara a aspectos de interoperabilidad y serialización de datos para comunicaciones remotas de Servicios Web, etc. También, el diseño y elección de tecnología para implementar las entidades, afectará en gran medida al rendimiento y eficiencia de nuestra capa de Dominio.

Opciones de tipos de datos/formato/tecnología:

- **Clases POCO**
  POCO significa '*Plain Old Clr Objects*', es decir, que implementaremos las entidades simplemente son clases sencillas de .NET, con variables miembro y propiedades para los atributos de la entidad. Esto puede hacerse manualmente o bien con la ayuda de generación de código de frameworks O/RM, como Entity Framework (EF) o NHibernate, que nos generen estas clases de forma automática, ahorrando mucho tiempo de desarrollo manual para sincronizarlo con el modelo entidad relación que estemos usando. La regla más importante de las clases POCO es que no deben tener dependencia alguna con otros componentes y/o clases. Deben ser simplemente clases .NET sencillas sin ninguna dependencia. Por ejemplo, una entidad normal de Entity Framework 1.0 no es una entidad POCO porque depende de clases base de las librerías de EF 1.0. Sin embargo, en EF 4.0 si es posible generar clases POCO completamente independientes a partir del modelo de EF.
  Estas clases POCO son apropiadas para arquitecturas N-*Layer* DDD.

- **Clases *Self-Tracking Entities* de EF 4.0 (IPOCO)**
  El concepto de clases IPOCO es prácticamente el mismo que el de clases POCO, todo lo que hemos dicho anteriormente se aplica de igual forma. La única diferencia radica en que en las entidades IPOCO se permite implementar interfaces concretos para aspectos que sean necesarios. Por ejemplo, las clases '*Self-Tracking'* de EF 4.0 (para poder realizar gestión de Concurrencia Optimista), son clases IPOCO, porque aunque son clases con código independiente, código de nuestro proyecto, sin embargo implementan un interfaz (o varios) requeridos por el sistema '*Self-Tracking'* de EF 4.0. Concretamente, se implementan los interfaces **IObjectWithChangeTracker y INotifyPropertyChanged.** Lo importante es que los interfaces que se implementen sean propios (código nuestro, como **IObjectWithChangeTracker** que es generado por las plantillas T4) o interfaces estándar de .NET Framework (como **INotifyPropertyChanged** que forma parte de .NET). Lo que no sería bueno es que se implementara un interfaz perteneciente a las propias librerías de Entity Framework o de otro O/RM, porque en este último caso tendríamos una dependencia directa con una tecnología y versión concreta de framework de persistencia de datos.
  Las clases IPOCO son también apropiadas para arquitecturas N-*Layer* DDD.

- **DataSets y DataTables (ADO.NET básico)**
  Los DataSets son algo parecido a bases de datos desconectadas en memoria que normalmente se mapean de una forma bastante cercana al propio esquema de base de datos. El uso de DataSets es bastante típico en implementaciones de .NET desde la versión 1.0, en un uso tradicional y normal de ADO.NET. Las ventajas de los DataSets es que son muy fáciles de usar, y en escenarios desconectados y aplicaciones muy orientadas a datos (CRUD) son muy productivos (Normalmente con un proveedor de ADO.NET para un SGBD concreto). También se puede hacer uso de 'LINQ to Datasets' para trabajar con ellos desde la sintaxis moderna de LINQ.
  Sin embargo, los DataSets tienen importantes desventajas, a considerar seriamente:

  1.- Los DataSets son muy poco interoperables hacia otras plataformas no Microsoft, como Java u otros lenguajes, por lo que aunque puedan ser serializados a XML, pueden ser un problema si se utilizan como tipos de datos en servicios web.

  2.- Aun en el caso de no requerirse la interoperabilidad con otras plataformas, los DataSets son objetos bastante pesados, especialmente cuando se serializan a XML y son utilizados en Servicios Web. El rendimiento de nuestros Servicios Web podría ser muy superior si se hace uso de clases propias (POCO/IPOCO) mucho más ligeras. Así pues, no se recomienda hacer uso de DataSets en comunicaciones a través de fronteras definidas por servicios web o incluso en comunicaciones inter-proceso (entre diferentes procesos .exe).

3.- Los O/RM (Entity Framework, etc.) no soportan/trabajan con DataSets.

4.- Los Datasets no están orientados a representar entidades puras de un Dominio y con su lógica de dominio incluida. El uso de Datasets no encaja en una Arqutiectura DDD porque realizaríamos un 'Dominio Anémico' al dejar separada la lógica de las entidades del dominio (en clases paralelas) de los datos de las entidades del dominio (en Datasets). Por eso, esta opción no encaja en DDD.

- **XML**
  Se trata de hacer uso simplemente de fragmentos de texto XML que contengan datos estructurados. Normalmente se suele hacer uso de esta opción (representar entidades del dominio con fragmentos XML) si la capa de presentación requiere XML o si la lógica del dominio debe trabajar con contenido XML que debe coincidir con esquemas concretos de XML. Otra ventaja del XML, es que al ser simplemente texto formateado, estas entidades serán completamente interoperables.
  Por ejemplo, un sistema en el que sería normal esta opción es un sistema de enrutamiento de mensajes donde la lógica enruta los mensajes basándose en nodos bien conocidos del documento XML. Hay que tener en cuenta que el uso y manipulación de XML puede requerir grandes cantidades de memoria en sistemas escalables (muchos usuarios simultáneos) y si el volumen de XML es importante, el acceso y proceso del XML puede llegar a ser también un cuello de botella cuando se procesa con APIs estándar de documentos XML.
  El gran problema de entidades basadas simplemente en XML es que no sería 'Domain Oriented' porque realizaríamos un 'Dominio Anémico' al dejar separada la lógica de las entidades del dominio de los datos de las entidades del dominio (XML). Por eso, esta opción no encaja en DDD.

Tabla 12.- Regla de Entidades del Dominio

Regla Nº: I5.	**Las entidades del dominio se implementarán como clases POCO o Self-Tracking Entities (IPOCO) de Entity Framework, generadas por las plantillas T4 o bien creadas manualmente.**

○ **Norma**

- Según las consideraciones anteriores, puesto que la presente Arquitectura Marco se trata de una Arquitectura Orientada al Dominio, y debemos conseguir la máxima independencia de los objetos del Dominio, las entidades del dominio se implementarán como clases POCO o 'Self-

Tracking' (IPOCO), normalmente generadas por las plantillas T4 de EF 4.0, para ahorrarnos mucho tiempo de implementación de dichas clases. Aunque crearlas manualmente es otra opción viable.

Ventajas de Entidades POCO e IPOCO

- Independencia de las entidades con respecto a librerías de tecnologías concretas.

- Son clases relativamente ligeras que ofrecen un buen rendimiento.

- Son la opción más adecuada para Arquitecturas *N-Layer* DDD.

¿Cuándo utilizar Entidades Self-Tracking de EF (IPOCO)?

- En la mayoría de aplicaciones en las que tenemos un control completo, se recomienda el uso de las **entidades 'Self-Tracking'** de EF (son **IPOCO**), porque son mucho más potentes que las entidades POCO. Las entidades 'Self-Tracking' nos ofrecen una gestión muy simplificada de concurrencia optimista en Aplicaciones N-Tier.

- **Las entidades '*self-tracking*' de EF (IPOCO) son adecuadas para aplicaciones N-Tier donde controlamos su desarrollo extremo a extremo. No son en cambio adecuadas para aplicaciones en las que no se quiere compartir los tipos de datos reales entre el cliente y el servidor, por ejemplo, aplicaciones puras SOA en las que controlamos solo uno de los extremos, bien el servicio o el consumidor. En estos otros casos en los que no se puede ni debe compartir tipos de datos (como SOA puro, etc.), se recomienda hacer uso de DTOs propios (*Data Transfer Objects*) en los servicios distribuidos (Servicios Web, etc.)**

¿Cuándo utilizar Entidades POCO?

- Por el contrario, si nuestra aplicación se trata de una aplicación o servicio con una fuerte orientación SOA, en los servicios distribuidos se usarán solo DTOs gestionando nosotros mismos casuísticas de concurrencia (*Optimistic Concurrency* gestionado por nosotros, etc.) y simplificando las entidades. En esos casos se recomienda el uso de entidades del dominio **POCO**, generadas por EF o por nosotros mismos. Las Entidades POCO ofrecen unas entidades muy simplificadas aunque nos ocasionará el tener que realizar un esfuerzo bastante mayor en la programación/implementación de nuestro sistema (Conversión de DTOS

a entidades del dominio, implementación manual de concurrencia optimista, etc.).

Referencias

POCO in the Entity Framework: **http://blogs.msdn.com/adonet/archive/2009/05/21/poco-in-the-entity-framework-part-1-the-experience.aspx**

Self-Tracking Entities in the Entity Framework: **http://blogs.msdn.com/efdesign/archive/2009/03/24/self-tracking-entities-in-the-entity-framework.aspx**

3.2.- Generación de entidades POCO/IPOCO con plantillas T4 de EF

Importante:
Aunque el concepto y situación de las entidades corresponde a la Capa del Dominio, **sin embargo, el momento de la generación de estas entidades se realiza con Visual Studio cuando estamos implementando la capa de infraestructura de persistencia de datos, creando el modelo entidad-relación de EF e implementando los repositorios. Por ello, el proceso de 'como generar las entidades POCO/IPOCO de EF' está explicado en el capítulo de implementación de Capa de Infraestructura de Persistencia de datos, pero situando dichas entidades en un assembly/proyecto perteneciente a la Capa de Dominio. Revisar dicho capítulo (título de generación de entidades con plantillas T4), si no se ha hecho hasta ahora.**

Finalmente, dispondremos de clases *custom* de entidades (clases POCO/IPOCO) generadas por EF, similares a la siguiente clase 'Self-Tracking' (IPOCO):

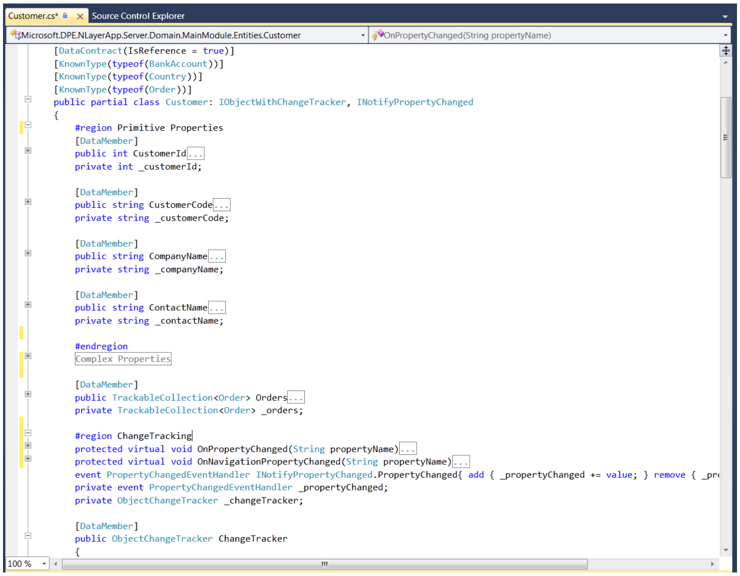


Figura 9.- Clases custom de entidades STE de EF

3.3.- Lógica del Dominio en las Clases de Entidades

En DDD es fundamental situar la lógica relativa a las operaciones internas de una entidad dentro de la clase de la propia entidad. Si las clases entidad las utilizásemos solamente como estructuras de datos y toda la lógica del domino estuviese separada y situada en los Servicios del Dominio, eso constituiría un anti-patrón llamado "Modelo de Dominio Anémico" (**Ver '*Anemic Domain Model*' definido originalmente por *Martin Fowler***). **Este punto es fundamental en DDD.**

Así pues, deberemos añadir a cada clase entidad la lógica de negocio/dominio relativa a la parte interna de datos de cada entidad. Si usamos entidades de EF (POCO o STE) generadas por las platillas T4, entonces, podemos añadirles lógica del dominio mediante clases parciales, como las que se pueden observar en el código siguiente llamado '*Clases custom-partial de entidades*'. Así por ejemplo, la siguiente clase parcial de '*BankAccount*', añadiría cierta lógica del dominio a la propia clase entidad del dominio. En concreto, la operación de cargo sobre la cuenta y comprobaciones de negocio necesarias antes de realizar un cargo en cuenta:

```
namespace Microsoft.Samples.NLayerApp.Domain.MainModule.Entities
{
    public partial class BankAccount
    {
        /// <summary>
        /// Deduct money to this account
        /// </summary>
        /// <param name="amount">Amount of money to deduct</param>
```

```
            public void ChargeMoney(decimal amount)
            {
                //Amount to Charge must be greater than 0. --> Domain logic.
                if (amount <= 0)
                    throw new ArgumentException(Messages.exception_InvalidArgument,
     "amount");
```

Comprobaciones/Validaciones de negocio

```
                //Account must not be locked, and balance must be greater than cero.
                if (!this.CanBeCharged(amount))
                    throw new
     InvalidOperationException(Resources.Messages.exception_InvalidAccountToBeCharged);

                //Charge means deducting money to this account. --> Domain Logic
                this.Balance -= amount;
            }
```

Lógica de Dominio/Negocio para proceso de Cargo en Cuenta Bancaria de entidad BankAccount

```
     ...
     ...
     ...
        }
     }
```

Figura 10.- Clases custom-partial de entidades

3.4.- Situación de Contratos/Interfaces de Repositorios en Capa de Dominio

Según explicamos en los capítulos teóricos de diseño DDD, son precisamente los interfaces de los repositorios lo único que se conocerá desde la capa de Dominio sobre los repositorios, y la propia instanciación de las clases *Repository* será realizada por el contenedor IoC elegido (en nuestro caso *Unity*). De esta forma, tendremos completamente desacoplada la capa de infraestructura de persistencia de datos y sus repositorios de las clases de la capa de Dominio.

Algo importante, sin embargo, es que los interfaces de los Repositorios deben situarse dentro de la Capa de Dominio, puesto que estamos hablando de los contratos que requiere el dominio para que una capa de infraestructura de repositorios pueda ser utilizada de forma desacoplada desde dicho dominio.

En el siguiente sub-esquema resaltamos donde se sitúan los contratos/interfaces de Repositorios dentro de la Capa de Dominio:

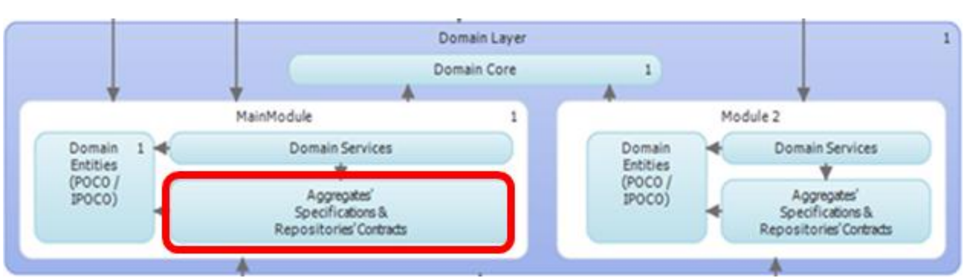


Figura 11.- Esquema de Contratos del Dominio

Así pues, estas abstracciones (interfaces) se definirán en nuestro ejemplo en el *namespace* siguiente dentro de la capa de Dominio:

```
Microsoft.Samples.NLayerApp.Domain.MainModule.Repositories.Contracts
```

De esta forma, podríamos llegar a sustituir completamente la capa de infraestructura de persistencia de datos, los repositorios 'en sí', su implementación, sin que afectara a la capa del Dominio, ni tener que cambiar dependencias ni hacer re-compilación alguna. Gracias a este desacoplamiento, podríamos hacer *mocking* de los repositorios y de una forma dinámica las clases de negocio del dominio instanciarían clases 'falsas' (*stubs* o *mocks*) sin tener que cambiar código ni dependencias, simplemente especificando al contenedor IoC que cuando se le pida que instancie un objeto para un interfaz dado, instancie una clase en lugar de otra (ambas cumpliendo el mismo interfaz, lógicamente).

Importante:
Aunque la situación de los contratos/interfaces de los repositorios debe estar situada en la Capa de Dominio por las razones anteriormente resaltadas, la implementación de ellos se hace, en el tiempo, simultáneamente a la propia implementación de los Repositorios, por lo que dicha implementación de interfaces de Repositorios está explicada con ejemplos de código en el capítulo de 'Implementación de Capa de Infraestructura de Persistencia de Datos'.

Tabla 13.- Situación de Contratos/Interfaces de Repositorios

Regla Nº: I6 — **Posicionar los contratos/interfaces de Repositorios en la Capa de Dominio.**

Norma

- Para poder maximizar el desacoplamiento entre la Capa de Dominio y la Capa de Infraestructura de Persistencia y Acceso a datos, es importante localizar los contratos/interfaces de repositorios como parte de la Capa de Dominio, y no en la propia Capa de Persistencia de Datos.

Referencias

Contratos de Repositorios en el Dominio – (Libro DDD de Eric Evans)

Un ejemplo de contrato/interfaz de Repositorio, dentro de la Capa del Dominio, puede ser el siguiente:

C#

```
namespace Microsoft.Samples.NLayerApp.Domain.MainModule.Contracts
{
   public interface IOrderRepository : IRepository
    {
     IEnumerable<Order> FindOrdersByDates(OrderDateSpecification
orderDateSpecification);

     IEnumerable<Order>
FindOrdersByShippingSpecification(OrderShippingSpecification
orderShippingSpecification);

     IEnumerable<Order> FindOrdersByCustomerCode(string customerCode);
    }
}
```

Namespace de Contratos de Repositorios está dentro de la Capa del Dominio

3.5.- Implementación de Servicios del Dominio

En el siguiente sub-esquema resaltamos donde se sitúan las clases de 'SERVICIOS del Dominio' dentro de la Capa de Dominio:

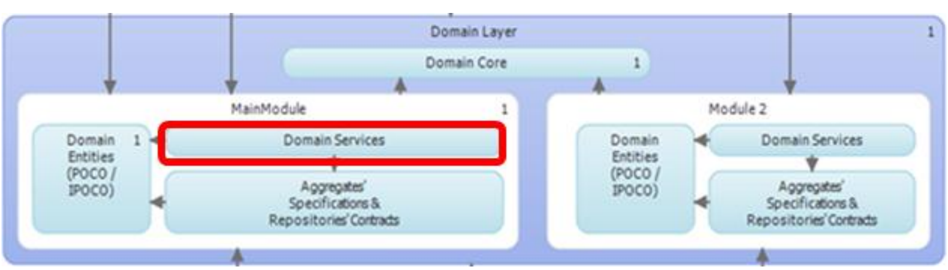


Figura 12.- Servicios del Dominio

Un SERVICIO es una operación o conjunto de operaciones ofrecidas como un interfaz que simplemente está disponible en el modelo.

La palabra "*Servicio*" del patrón SERVICIO precisamente hace hincapié en lo que ofrece: "*Qué puede hacer y qué acciones ofrece al cliente que lo consume, y enfatiza la relación con otros objetos del Dominio (Englobando varias Entidades, en algunos casos)*".

Normalmente implementaremos las clases de SERVICIOS como simples clases .NET con métodos donde se implementan las diferentes posibles acciones relacionadas con una o varias entidades del Dominio. En definitiva, implementación de acciones como métodos.

Las clases de SERVICIOS deben encapsular y aislar a la capa de infraestructura de persistencia de datos. Es en estos componentes de negocio donde deben implementarse

todas reglas y cálculos de negocio que no sean internos a las propias ENTIDADES, como por ejemplo, operaciones complejas/globales que impliquen el uso de varios objetos de entidades, así como validaciones de negocio de datos requeridos para un proceso.

3.5.1.- SERVICIOS del Dominio como coordinadores de procesos de Negocio

Como se explicó en más detalle a nivel de diseño en el capítulo de Arquitectura y diseño de la Capa de Dominio, **las clases SERVICIO son fundamentalmente coordinadores de procesos de negocio normalmente abarcando diferentes conceptos y ENTIDADES relacionadas con escenarios y casos de uso completos.**

Por ejemplo, un Servicio del Dominio sería una clase que coordine una operación que englobe a diferentes entidades, e incluso operaciones de otros Servicios relacionados.

El siguiente código es un ejemplo de una clase de SERVICIO del Dominio que pretende coordinar una operación de negocio:

Namespace **de los Servicios del Dominio en un módulo concreto**

```
C#
…
…
namespace Microsoft.Samples.NLayerApp.Domain.MainModule.Services
{
```

Servicio del Dominio — **Contrato/Interfaz a cumplir**

```
    public class TransferService : ITransferService
    {
        public void PerformTransfer(BankAccount originAccount,
BankAccount destinationAccount, decimal amount)
        {
            //Domain Logic
            //Process: Perform transfer operations to in-memory Domain-
Model Objects
            // 1.- Charge money to origin acc
            // 2.- Credit money to destination acc
            // 3.- Annotate transfer to origin account

            //Number Accounts must be different
            if (originAccount.BankAccountNumber !=
destinationAccount.BankAccountNumber)
            {
                //1. Charge to origin account (Domain Logic)
                originAccount.ChargeMoney(amount);
```

Cargar en Cuenta

```
                //2. Credit to destination account (Domain Logic)
                destinationAccount.CreditMoney(amount);
```

Abonar en Cuenta

```
                //3. Anotate transfer to related origin account
```

```
                originAccount.BankTransfersFromThis.Add(new
BankTransfer()
                {
                    Amount = amount,
                    TransferDate = DateTime.UtcNow,
                    ToBankAccountId = destinationAccount.BankAccountId
                });
            }
            else
                throw new
InvalidOperationException(Resources.Messages.exception_InvalidAccountsFo
rTransfer);
        }
    }
}
```

Anotar operación

Como se puede observar, el código anterior de Servicio del Dominio es muy limpio y solo relativo a la lógica de negocio y datos de negocio. No hay operaciones de ‘fontanería de la aplicación’ como podría ser el uso de Repositorios, Unit of work, creación de transacciones, etc.

En los métodos de los Servicios del Dominio simplemente debemos interactuar con la lógica ofrecida por las entidades que entran en juego. En el ejemplo anterior llamamos a métodos (*ChargeMoney*(), *CreditMoney*(), etc.) que pertenecen a las propias entidades (es un modelo DDD, no es un ‘*Anemic Domain Model*’ porque tenemos lógica en las propias entidades).

Recalcar que, normalmente, **en la ejecución de los métodos de un Servicio del Dominio, todas la operaciones las hacemos solamente contra los objetos/entidades que están en memoria, y cuando acaba la ejecución de nuestro método de Servicio de Dominio, simplemente habremos modificado datos de Entidades y/u Objetos Valor de nuestro modelo de EF, pero todos esos cambios estarán todavía solo en la memoria del servidor (Entidades del contexto de EF).** La persistencia de dichos objetos y cambios en datos que realiza nuestra lógica no se realizará hasta que lo coordinemos/ordenemos desde nuestra Capa superior de Aplicación que será la que invoque a los Repositorios dentro de una lógica de aplicación compleja (UoW y transacciones).

Es también dicha capa superior, la Capa de Aplicación, la que normalmente llamará a los servicios del Dominio, proporcionándole las entidades necesarias habiendo realizado sus correspondientes consultas mediante Repositorios. Y finalmente esa capa de aplicación será también la que coordine la persistencia en almacenes y bases de datos.

Importante:
Saber contestar a esta pregunta, es fundamental:
'¿Qué código implemento en los Servicios de la Capa de Dominio'
La contestación es:
'Solo operaciones de negocio que discutiríamos con un Experto del Dominio o un usuario final'.
Con un experto del dominio no hablaríamos de 'fontanería de la aplicación', cómo crear transacciones, UoW, uso de Repositorios, persistencia, etc., por eso, todo lo que sea coordinación, pero no sea pura lógica del dominio, debería normalmente situarse en la Capa de Aplicación, para no 'ensuciar' la lógica del Dominio.

3.6.- Patrón ESPECIFICACION (SPECIFICATION pattern)

Como se explicó en el capítulo de lógica de la Capa de Dominio, el enfoque del patrón **ESPECIFICACION** consiste en **separar la sentencia de qué tipo de objetos deben ser seleccionados en una consulta del propio objeto que realiza la selección**. El objeto 'Especificación' tendrá una responsabilidad clara y limitada que deberá estar separada y desacoplada del objeto de Dominio que lo usa.

Este patrón está explicado a nivel lógico y en detalle, en un '*paper*' conjunto de *Martin Fowler* y *Eric Evans*: **http://martinfowler.com/apsupp/spec.pdf**

Así pues, la idea principal es que la sentencia de 'que' datos candidatos debemos obtener debe estar separada de los propios objetos candidatos que se buscan y del mecanismo utilizado para buscarlos.

3.6.1.- Uso del patrón SPECIFICATION

El uso del patrón *specification* se realizará normalmente desde la capa de aplicación donde definimos las consultas lógicas que se quieren realizar, pero lo tendremos desacoplado con respecto a la implementación real de dichas consultas lógicas que estará en la capa de infraestructura de persistencia y acceso a datos.

A continuación mostramos código donde usamos una especificación.

```
… //Application Layer
…
                    Método con uso sencillo de ESPECIFICACION

public Customer FindCustomerByCode(string customerCode)
{
    //Create specification
    CustomerCodeSpecification spec = new
CustomerCodeSpecification(customerCode);

    return _customerRepository.FindCustomer(spec);
}
…
…                   Método con uso complejo de ESPECIFICACION

public List<Customer> FindPagedCustomers(int pageIndex, int pageCount)
{
   //Create "enabled variable" transform adhoc execution plan in prepared plan
   bool enabled = true;
   Specification<Customer> onlyEnabledSpec = new
DirectSpecification<Customer>(c => c.IsEnabled == enabled);

   return _customerRepository.GetPagedElements(pageIndex, pageCount, c =>
c.CustomerCode, onlyEnabledSpec, true)
                                      .ToList();
}
```

3.6.2.- Implementación del patrón SPECIFICATION

Sin embargo, la implementación elegida por nosotros difiere en parte del patrón lógico original definido por MF y EE, debido a la mayor potencia de lenguaje que se nos ofrece en .NET, como por ejemplo los árboles de expresiones, donde podemos obtener mucho más beneficio que si trabajamos solamente con especificaciones para objetos en memoria, como MF y EE lo definieron originalmente.

La definición original de este patrón, mostrada en el diagrama UML siguiente, muestra que se trabaja con objetos y/o colecciones de objetos que deben satisfacer una especificación.

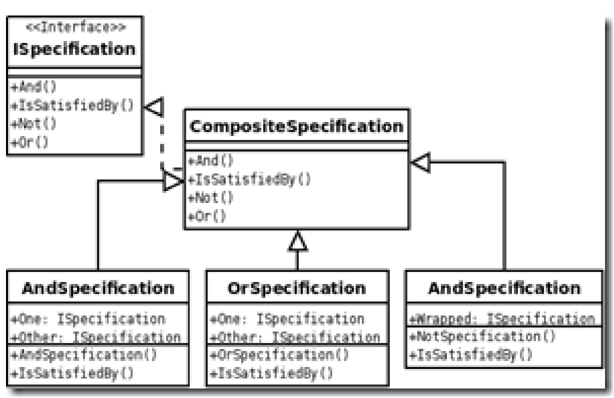


Figura 13.- Diagrama UML de Especificaciones Compuestas

Esto es precisamente lo que comentábamos que no tiene sentido en una implementación avanzada con .NET y EF (u otro ORM) donde podemos trabajar con consultas que directamente trabajarán contra la base de datos en lugar de objetos en memoria, como pre-supone originalmente el patrón SPECIFICATION.

La razón principal de la afirmación anterior viene de la propia definición del patrón, la cual implica trabajar con objetos directamente en memoria puesto que el método **IsSatisfiedBy()** tomaría una instancia del objeto en el cual queremos comprobar si cumple un determinado criterio y devolver **true** o **false** según se cumpla o no, algo que por supuesto no deseamos por la sobrecarga que esto implicaría. Por todo esto podríamos modificar un poco nuestra definición de ESPECIFICACION para que en vez de devolver un booleano negando o afirmando el cumplimiento de una especificación determinada, devolvamos una "*expression*" con el criterio a cumplir.

En el siguiente fragmento de código tendríamos un esqueleto de nuestro contrato base con esta ligera modificación:

C#

***Esqueleto/Interfaz* de nuestro contrato base**

```
public interface ISpecification<TEntity>
        where TEntity : class,new()
{
    /// <summary>
    /// Check if this specification is satisfied by a
    /// specific lambda expression
    /// </summary>
    /// <returns></retur
    Expression<Func<TEntity, bool>> SatisfiedBy();
}
```

***Utilizamos* SatisfiedBy() en lugar del original IsSatisfiedBy()**

Llegados a este punto podríamos decir que ya tenemos la base y la idea de lo que queremos construir, ahora, solamente falta seguir las propias normas y guías de este patrón empezándonos a crear nuestras especificaciones directas o "*hard coded specifications*" y nuestras especificaciones compuestas, al estilo And, Or, etc.

Tabla 14.- Objetivo de implementación de patrón ESPECIFICACION

Objetivo de implementación de patrón ESPECIFICACION
En definitiva, buscamos una forma elegante en la que, manteniendo el principio de separación de responsabilidades y teniendo en cuenta que una ESPECIFICACION es un concepto de negocio (un tipo especial de búsqueda perfectamente explícito), se puedan hacer consultas distintas en función de parámetros usando conjunciones o disyunciones de expresiones.

Podríamos declarar especificaciones como la siguiente:

Especificacion para obtener Pedidos dependiendo de luna Dirección de Envío

Constructor con valores requeridos por la Especificación. Tener en cuenta que no tiene sentido utilizar DI/IoC para instanciar un objeto de Especificación

El método SatisfiedBy() devuelve una Expresión Lambda de Linq

```
C#

/// <summary>
/// AdHoc specification for finding orders
/// by shipping values
/// </summary>
public class OrderShippingSpecification
    : Specification<Order>
{
    string _ShippingName = default(String);
    string _ShippingAddress = default(String);
    string _ShippingCity = default(String);
    string _ShippingZip = default(String);

    public OrderShippingSpecification(string shippingName, string
shippingAddress, string shippingCity, string shippingZip)
    {
        _ShippingName = shippingName;
        _ShippingAddress = shippingAddress;
        _ShippingCity = shippingCity;
        _ShippingZip = shippingZip;
    }

    public override System.Linq.Expressions.Expression<Func<Order,
bool>> SatisfiedBy()
    {
        Specification<Order> beginSpec = new
TrueSpecification<Order>();

if (_ShippingName != null)
            beginSpec &= new DirectSpecification<Order>(o =>
o.ShippingName != null && o.ShippingName.Contains(_ShippingName));

if (_ShippingAddress != null)
            beginSpec &= new DirectSpecification<Order>(o =>
o.ShippingAddress != null &&
o.ShippingAddress.Contains(_ShippingAddress));

if (_ShippingCity != null)
            beginSpec &= new DirectSpecification<Order>(o =>
o.ShippingCity != null && o.ShippingCity.Contains(_ShippingCity));

if (_ShippingZip != null)
            beginSpec &= new DirectSpecification<Order>(o =>
o.ShippingZip != null && o.ShippingZip.Contains(_ShippingZip));

        return beginSpec.SatisfiedBy();

    }

}
```

Fíjese como la especificación anterior, *OrderShippingSpecification*, nos proporciona un mecanismo para saber el criterio de los elementos que deseamos buscar, pero para nada sabe acerca de quien realizará la operación de búsqueda de los mismos. Además de esta clara separación de responsabilidades, la creación de estos elementos, también nos ayuda a dejar

perfectamente claras operaciones importantes del dominio, como por ejemplo, tipos de criterios de búsqueda, que de otra forma tendríamos desperdigados por distintas partes de código y por lo tanto serían más difíciles y costosos de modificar. Para terminar, otra de las ventajas de las especificaciones, tal y como están propuestas viene de la posibilidad de realizar operaciones lógicas sobre las mismas, dándonos, en definitiva, un mecanismo para realizar consultas dinámicas en Linq, de una forma sencilla.

3.6.3.- Especificaciones compuestas por operadores AND y OR

Es seguro que existe más de una aproximación para implementar estos operadores pero nosotros hemos optado por implementarlo con el patrón VISITOR para evaluar las expresiones (ExpressionVisitor:
http://msdn.microsoft.com/en-us/library/system.linq.expressions.expressionvisitor(VS.100).aspx).

Lo que necesitamos es la siguiente clase que nos haga una recomposición de las expresiones en vez de un **InvocationExpression** (que no es compatible con EF 4.0).

Esta clase de apoyo es la siguiente:

C#

```
/// <summary>
/// Extension method to add AND and OR with rebinder parameters
/// </summary>
public static class ExpressionBuilder
{
    public static Expression<T> Compose<T>(this Expression<T> first,
Expression<T> second, Func<Expression, Expression, Expression> merge)
    {
        // build parameter map (from parameters of second to
parameters of first)
        var map = first.Parameters.Select((f, i) => new { f, s =
second.Parameters[i] }).ToDictionary(p => p.s, p => p.f);

        // replace parameters in the second lambda expression with
parameters from the first
        var secondBody = ParameterRebinder.ReplaceParameters(map,
second.Body);
        // apply composition of lambda expression bodies to
parameters from the first expression
        return Expression.Lambda<T>(merge(first.Body, secondBody),
first.Parameters);
    }
    public static Expression<Func<T, bool>> And<T>(this
Expression<Func<T, bool>> first, Expression<Func<T, bool>> second)
    {
        return first.Compose(second, Expression.And);
    }
    public static Expression<Func<T, bool>> Or<T>(this
Expression<Func<T, bool>> first, Expression<Func<T, bool>> second)
    {
        return first.Compose(second, Expression.Or);
    }

}
```

Constructor de EXPRESIONES

La definición completa por lo tanto de una especificación AND nos queda como sigue:

C#

```
/// <summary>
/// A logic AND Specification
/// </summary>
/// <typeparam name="T">Type of entity that checks this
specification</typeparam>
public class AndSpecification<T> : CompositeSpecification<T>
   where T : class,new()
{

    private ISpecification<T> _RightSideSpecification = null;
    private ISpecification<T> _LeftSideSpecification = null;

    /// <summary>
    /// Default constructor for AndSpecification
    /// </summary>
    /// <param name="leftSide">Left side specification</param>
    /// <param name="rightSide">Right side specification</param>
    public AndSpecification(ISpecification<T> leftSide,
ISpecification<T> rightSide)
    {
        if (leftSide == (ISpecification<T>)null)
            throw new ArgumentNullException("leftSide");

        if (rightSide == (ISpecification<T>)null)
            throw new ArgumentNullException("rightSide");

        this._LeftSideSpecification = leftSide;
        this._RightSideSpecification = rightSide;
    }

    /// <summary>
    /// Left side specification
    /// </summary>
    public override ISpecification<T> LeftSideSpecification
    {
        get { return _LeftSideSpecification; }
    }

    /// <summary>
    /// Right side specification
    /// </summary>
    public override ISpecification<T> RightSideSpecification
    {
        get { return _RightSideSpecification; }
    }
    public override Expression<Func<T, bool>> SatisfiedBy()
    {
        Expression<Func<T, bool>> left =
_LeftSideSpecification.SatisfiedBy();
        Expression<Func<T, bool>> right =
_RightSideSpecification.SatisfiedBy();

        return (left.And(right));
    }
}
```

Especificacion AND

El método SatisfiedBy() requerido por nuestro patrón SPECIFICATION

Dentro de la jerarquía de especificaciones que se propone en el documento de Evans y Fowler podemos encontrar desde la especificación NOT hasta una base para LeafSpecifications que tendríamos que construir.

3.7.- Implementación de pruebas en la capa del dominio

Al igual que cualquiera de los demás elementos de una solución, nuestra Capa de Dominio es otra superficie que también debería estar cubierta por un conjunto de pruebas y, por supuesto, cumplir los mismos requisitos que se le exigen en el resto de capas o de partes de un proyecto. Dentro de esta capa los principales puntos que deben disponer de una buena cobertura de código son las entidades y la sub capa de servicios del dominio.

Respecto a las entidades debemos crear pruebas para los métodos de negocio internos de las mismas puesto que el resto de código es generado de forma automática por las plantillas T4 de Entity Framework tal y como se ha comentado en puntos anteriores. El caso de los servicios del dominio es distinto ya que todo el código es adhoc y por lo tanto deberíamos disponer de pruebas para cada uno de los elementos desarrollados. Para cada uno de los módulos de la solución debe agregarse un proyecto de pruebas de la capa de Dominio. Así, si disponemos del módulo *MainModule* deberemos tener de un proyecto de pruebas *Domain.MainModule.Tests* dónde tendremos el conjunto de pruebas tanto de entidades como de servicios.

En el siguiente ejemplo de código pueden verse algunas de las pruebas de la entidad del dominio *BankAccount*:

```
C#
[TestClass()]
public class BankAccountTest
{

    [TestMethod()]
    public void CanTransferMoney_Invoke_Test()
    {
        //Arrange
        BankAccount bankAccount = new BankAccount()
        {
            BankAccountId = 1,
            Balance = 1000M,
            BankAccountNumber = "A001",
            CustomerId = 1,
            Locked = false
        };

        //Act
        bool canTransferMoney = bankAccount.CanTransferMoney(100);

        //Assert
        Assert.IsTrue(canTransferMoney);
```

```
        }

        [TestMethod()]
        public void CanTransferMoney_ExcesibeAmountReturnFalse_Test()
        {
            //Arrange
            BankAccount bankAccount = new BankAccount()
            {
                BankAccountId = 1,
                Balance = 100M,
                BankAccountNumber = "A001",
                CustomerId = 1,
                Locked = false
            };

            //Act
            bool canTransferMoney = bankAccount.CanTransferMoney(1000);

            //Assert
            Assert.IsFalse(canTransferMoney);
        }
        [TestMethod()]
        public void CanTransferMoney_LockedTruetReturnFalse_Test()
        {
            //Arrange
            BankAccount bankAccount = new BankAccount()
            {
                BankAccountId = 1,
                Balance = 1000M,
                BankAccountNumber = "A001",
                CustomerId = 1,
                Locked = true
            };

            //Act
            bool canTransferMoney = bankAccount.CanTransferMoney(100);

            //Assert
            Assert.IsFalse(canTransferMoney);
        }
    }
```

Tabla 15.- Pruebas en la capa de dominio

Regla Nº: I7. **Implementación de pruebas en la capa del Dominio.**

○ **Recomendación**

- Agregar la posibilidad de que las pruebas de la capa del dominio se puedan ejecutar de forma aislada a cualquier dependencia, como por ejemplo una

base de datos. Esto permite que las pruebas se ejecuten más rápidamente y por lo tanto el desarrollador no tendrá inconvenientes en ejecutar un conjunto de las mismas en cada cambio de código.

- Verificar que todas las pruebas son repetibles, es decir, que dos ejecuciones secuenciales de una misma prueba devuelven el mismo resultado, sin necesidad de realizar un paso previo.

- Evitar excesivo código de preparación y limpieza de las pruebas puesto que podría afectar a la legibilidad de las mismas

Referencias

Unit Test Patterns: http://xunitpatterns.com

CAPÍTULO

6

Capa de Aplicación

1.- CAPA DE APLICACION

Esta Capa de Aplicación, siguiendo las tendencias de Arquitectura DDD, debe ser una Capa delgada que coordina actividades de la Aplicación como tal, pero es fundamental que no incluya lógica de negocio ni tampoco por lo tanto estados de negocio/dominio. Si puede contener estados de progreso de tareas de la aplicación.

Los SERVICIOS que viven típicamente en esta capa (recordar que el patrón SERVICIO es aplicable a diferentes capas de la Arquitectura), son servicios que normalmente coordinan SERVICIOS de otras capas de nivel inferior.

El caso más normal de un Servicio de Aplicación es un Servicio que coordine toda la 'fontanería' de nuestra aplicación, es decir, orquestación de llamadas a Servicios del Dominio y posteriormente llamadas a Repositorios para realizar la persistencia, junto con creación y uso de UoW, transacciones, etc.

Otro caso más colateral sería un SERVICIO de la capa APLICACIÓN encargado de recibir órdenes de compra de un Comercio Electrónico en un formato concreto XML. Se puede encargar en la capa de APLICACIÓN de reformatear/reconstruir dichas Órdenes de Compra a partir de dicho XML original recibido y convertirlas en objetos de ENTIDADES del Modelo de Dominio. Este ejemplo es un caso típico de APLICACIÓN, necesitamos realizar una conversión de formatos, es una necesidad de la aplicación, no es algo que forme parte de la lógica del Dominio, por lo tanto no está dentro de la Capa de Dominio sino de esta capa de Aplicación.

En definitiva, situaremos en los SERVICIOS de la Capa de Aplicación toda la coordinación necesaria para realizar operaciones/escenarios completos, pero de las tareas de aplicación que nunca hablaríamos con un experto del dominio (usuario final), llamado informalmente coordinación de la 'fontanería' de la aplicación. También la capa de Aplicación suele servirnos de 'Capa Fachada' (*Façade Layer*) hacia quien consume los componentes de servidor (Capas de presentación u otros servicios remotos).

2.- ARQUITECTURA Y DISEÑO LÓGICO DE LA CAPA DE APLICACIÓN

En el siguiente diagrama se muestra cómo encaja típicamente esta capa de Aplicación dentro de nuestra arquitectura *N-Capas Orientada al Dominio*:

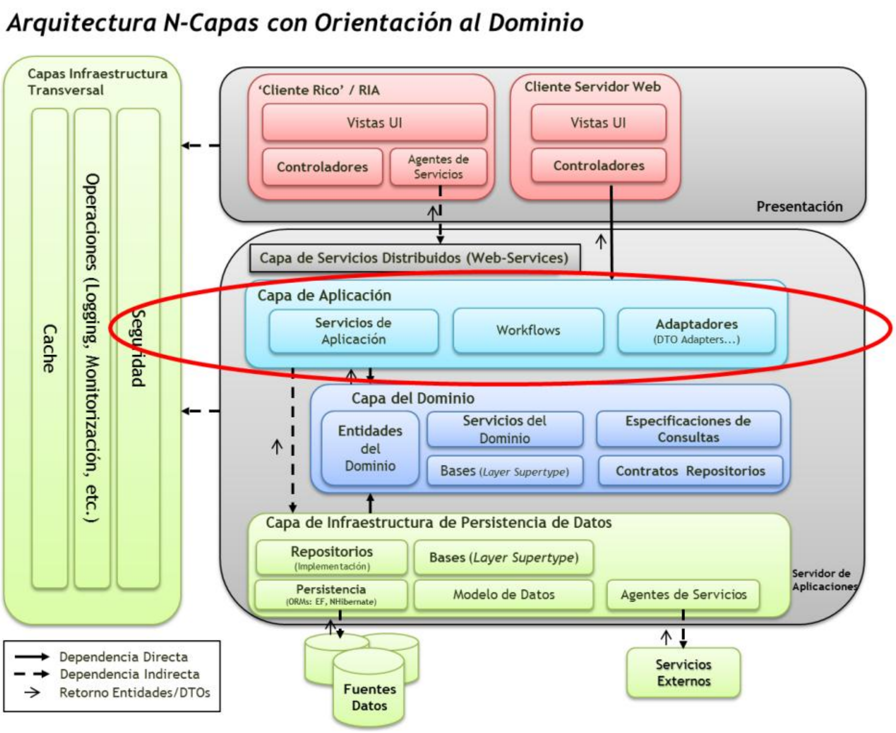


Figura 1.- Situación de Capa de Aplicación en Arquitectura N-Capas DDD

La Capa de Aplicación, por lo tanto, define las tareas que se supone debe hacer el software, como tal, lo cual normalmente está ligado finalmente a realizar llamadas a la Capa de Dominio e Infraestructura. Sin embargo, las tareas que sean exclusivas de la aplicación y no del Dominio (p.e. coordinación de llamadas a Repositorios para persistir datos en base de datos, conversiones de datos, ofrecer una granularización mayor de interfaces para mejorar el rendimiento en las comunicaciones, implementación de Adaptadores DTO para realizar conversiones de datos, etc.) son las tareas que debemos coordinar en esta capa.

Los elementos a incluir en la Capa de Aplicación pueden ser:

- **Servicios de Aplicación** (Es el elemento más común en esta capa).
- **Workflows** (Flujos de trabajo para ejecuciones largas de procesos).
- **Adaptadores/Conversores** (P.e. Conversores de DTO a entidades del Dominio)

Tabla 1.- Guía de Arquitectura Marco

Regla Nº: D17.

Se diseñará e implementará una Capa de Aplicación para coordinación de tareas relativas a requerimientos técnicos propios de la Aplicación.

Normas

- La lógica de Aplicación no deberá incluir ninguna lógica del Dominio, solo tareas de coordinación relativas a requerimientos técnicos de la aplicación, como coordinación de llamadas a Repositorios para persistir datos, conversiones de formatos de datos de entrada a entidades del Dominio, y en definitiva, llamadas a componentes de Infraestructura para que realicen tareas complementarias de aplicación.

- Nunca deberá poseer estados que reflejen la situación de los procesos de negocio, sin embargo si puede disponer de estados que reflejen el progreso de una tarea del software.

Ventajas del uso de la Capa de Aplicación

→ Cumplimos el principio de "*Separation of Concerns*", es decir, aislamos a la Capa de Dominio de tareas/requerimientos propios del software, tareas de 'fontanería' que realmente no es lógica del negocio, sino aspectos de integración tecnológica, coordinación de la persistencia de datos, formatos de datos, optimización del rendimiento, etc.

Referencias

Capa 'Application'. Por Eric Evans en su libro DDD.

2.1.- Proceso de diseño de capa de Aplicación

Al diseñar las capas de componentes de servidor, tenemos que tener en cuenta los requerimientos de diseño de la aplicación. En esta sección explicamos brevemente las principales actividades relacionadas con el diseño de los componentes normalmente situados en la Capa de Aplicación. Se deben realizar las siguientes actividades clave en cada una de las áreas cuando realizamos el diseño de las capas de negocio:

1.- Crear un diseño general de las capas de aplicación:

a. Identificar los consumidores de la capa de aplicación.

b. Determinar cómo se expondrán las capas de aplicación.

c. Determinar los requerimientos de seguridad en la capa de aplicación.

d. Determinar los requerimientos y la estrategia de validación en la capa de aplicación.

e. Determinar la estrategia de Cache.

f. Determinar el sistema de gestión de excepciones de la capa de aplicación.

2.- Diseño de los componentes de lógica de aplicación (Coordinación).

a. Identificar los componentes del dominio que coordinaremos desde la capa de aplicación.

b. Tomar decisiones sobre localización, acoplamiento e interacciones de los componentes de negocio.

c. Elegir un soporte transaccional adecuado.

d. Identificar como implementar la coordinación de las reglas de negocio.

  a. Directamente en código, con Servicios de aplicación.

  b. Orientación a eventos del dominio (*Event Sourcing* y EDA)

  c. CQRS (*Command & Query Responsability Segregation*), commandos, eventos, etc.

  d. Workflows (Escenarios con ejecuciones largas)

e. Identificar patrones de diseño de la capa de Aplicación que se ajusten a los requerimientos.

2.2.- La importancia del desacoplamiento de la Capa de Aplicación con respecto a Infraestructura

En el diseño de la capa de Aplicación y del resto de capas más internas, debemos asegurarnos de que implementamos un mecanismo de desacoplamiento entre los objetos de dichas capas, esto permitirá que en escenarios de aplicaciones de negocio con un gran volumen de componentes de lógica del dominio (reglas de negocio) y un alto nivel de acceso a fuentes de datos, aun así podamos soportar un fuerte ritmo de cambios en dichas reglas de negocio (**el desacoplamiento entre capas mediante contratos e interfaces e incluso yendo más allá con DI (*Dependency Injection*) e IoC (*Inversion of Control*), aportan una muy buena mantenibilidad del sistema**), gracias a tener bien localizadas y aisladas las reglas/lógica del dominio, ya que no hay dependencias directas entre las capas de alto nivel. Para mayor información sobre conceptos genéricos de Inversión de Control e Inyección de Dependencias, ver el capítulo inicial global de la Arquitectura N-Layer.

3.- COMPONENTES DE LA CAPA DE APLICACIÓN

La Capa de Aplicación puede incluir diferentes tipos de componentes, pero ciertamente, el tipo de componente principal será el de SERVICIO de Aplicación, como explicamos a continuación.

3.1.- Servicios de Aplicación

El SERVICIO de Aplicación es otro tipo más de Servicio, cumpliendo con las directrices de su patrón (*SERVICE pattern*). Básicamente deben ser objetos sin estados que coordinen ciertas operaciones, en este caso operaciones y tareas relativas a la Capa de Aplicación (Tareas requeridas por el software/aplicación, no por la lógica del Dominio).

Otra función de los Servicios de Aplicación es encapsular y aislar a la capa de infraestructura de persistencia de datos. Así pues, en la capa de aplicación realizaremos la **coordinación** de transacciones y persistencia de datos (solo la coordinación o llamada a los Repositorios), validaciones de datos y aspectos de seguridad como requerimientos de autenticación y autorización para ejecutar componentes concretos, etc.

También los SERVICIOS deben ser normalmente el único punto o tipo de componente de la arquitectura por el que se acceda a las clases de infraestructura de persistencia de datos (*Repositorios*) desde capas superiores. No debería, por ejemplo,

poder invocar directamente desde la Capa de Presentación (p.e. Web) objetos Repositorios de la Capa de Persistencia y Acceso a Datos.

Como se puede observar en el diagrama, la interacción entre los diferentes objetos de las capas de la aplicación normalmente comenzará en un Servicio de Aplicación, el cual servirá de concentrador o *hub* de los diferentes tipos de acciones de la aplicación.

Ejemplo de interacción entre objetos de Servicios de Aplicación, Servicios del Dominio y Repositorios de Persistencia de datos

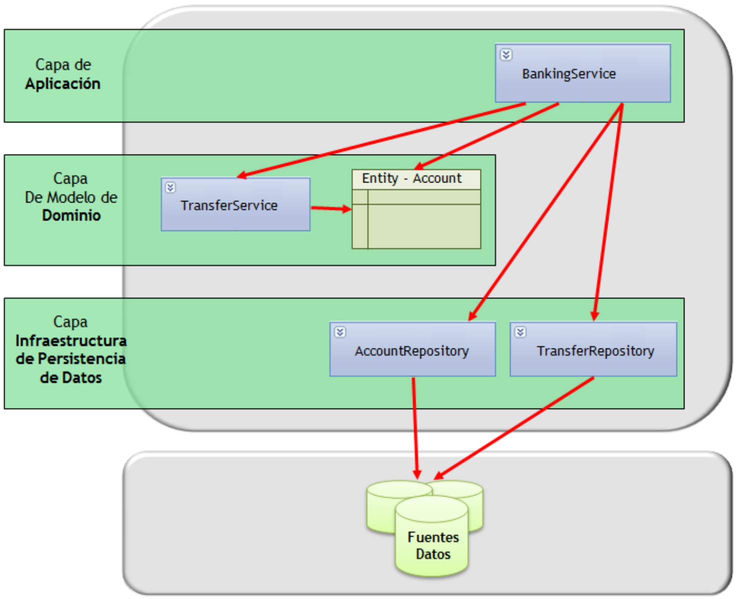


Figura 2.- ejemplo de interacción entre objetos de diferentes capas

El aspecto fundamental de esta capa es no mezclar requerimientos de Software (coordinación de la persistencia, conversiones a diferentes formatos de datos, optimizaciones, Calidad de Servicio, etc.) con la Capa de Dominio que solo debe contener pura lógica de Negocio.

En el diagrama UML de secuencia que mostramos a continuación, se pueden observar los objetos de la Capa de Aplicación (el Servicio que origina la transferencia bancaria en la aplicación), los objetos del Dominio y como posteriormente desde el Servicio de aplicación se llama a los objetos Repositorio y UoW.

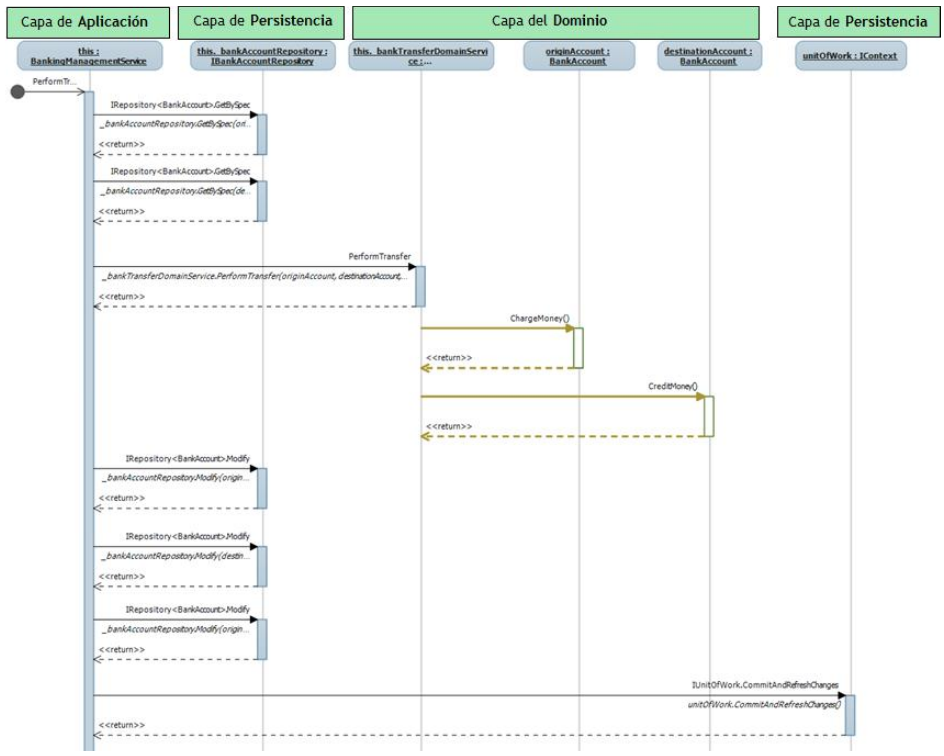


Figura 3.- Diagrama de secuencia

En esta interacción entre objetos, solamente las llamadas a métodos *ChargeMoney()* y *CreditMoney()* son puramente de negocio/Dominio. El resto de interacciones son aspectos necesarios de la aplicación (consulta de datos de cada cuenta y persistencia de datos mediante Repositorios; uso de UoW etc.), y por lo tanto, acciones a coordinar desde la Capa de Aplicación.

Regla Nº: D18.	**Clases de SERVICIOS como únicos responsables de interlocución con clases de la capa de Infraestructura de persistencia de datos (Clases *Repository*)**

○ **<u>Recomendación</u>**

- Es recomendable que las clases de SERVICIOS de aplicación sean las únicas responsables (interlocutores o vía de acceso) con las clases

Repository de la capa inferior de Infraestructura. Por ejemplo, no se debe de poder acceder a una clase Repository directamente desde capa de Servicios-Web o Presentación-Web. Incluso normalmente desde una clase del dominio tampoco instanciaremos Repositorios, aunque esta última recomendación puede tener excepciones.

- De esta forma, estaremos seguros de que la lógica de Aplicación relativa a conjuntos y colecciones de entidades y Repositorios se aplica en la Capa de Aplicación y que no estamos saltándonos dicha lógica y validaciones, cosa que pasaría si se accede directamente a las clases de Repositorios.

Nota1: Es factible implementar coordinación de Repositorios, UoW, transacciones, etc. dentro de los propios objetos/servicios del Dominio y de hecho hay muchas implementaciones de arquitecturas N-Capas, incluso siendo DDD, que lo realizan así. El situar la coordinación de Repositorios en una u otra capa es simplemente por razones de preferencias en el diseño. Dejando estos aspectos en la Capa de Aplicación (como preferimos hacer), la Capa del Dominio queda mucho más limpia y simplificada, mostrando solamente lógica del dominio.

Nota2: Adicionalmente y aunque como norma solo se consuman los Repositorios desde la capa de Aplicación, es factible también hacer excepciones, y en casos donde sea necesario poder realizar consultas invocando a Repositorios desde dentro de Servicios del Dominio. Pero esta excepción deberíamos minimizarla al máximo, por homogeneidad en nuestros desarrollos.

Regla Nº: D19. **No implementar código de persistencia/acceso a datos en los Servicios de Aplicación**

○ **<u>Norma</u>**

- No implementar nunca código de persistencia o acceso a datos (como 'LinQ to Entities', 'LinQ to SQL', ADO.NET, etc.) ni código de sentencias SQL o nombres de procedimientos almacenados, directamente en los métodos de las clases de Aplicación. Para el acceso a datos, se deberá invocar solamente

a clases y métodos de la capa de Infraestructura (Invocar a las clases *Repository*).

Referencias

Principio "Separation of Concerns"
http://en.wikipedia.org/wiki/Separation_of_concerns

Regla Nº: D20.

Implementar patrón "Capa Super-tipo" (*Layer Supertype*)

Recomendación

- Es usual y muy útil disponer de 'clases base' de cada capa para agrupar y reutilizar métodos comunes que no queremos tener duplicados en diferentes partes del sistema. A este sencillo patrón se le llama "*Layer SuperType*".

- Si bien es cierto que debe implementarse solo si es necesario (YAGNI).

Referencias

Patrón '*Layer Supertype*'. Por Martin Fowler.
http://martinfowler.com/eaaCatalog/layerSupertype.html

3.2.- Desacoplamiento entre SERVICIOS de APLICACION y REPOSITORIOS

Al desacoplar todos los objetos con dependencias mediante **IoC y DI, también quedará desacoplada la capa de lógica de aplicación con respecto a las capas inferiores como los Repositorios (Pertenecientes a la Capa de Infraestructura de Persistencia de Datos). De esta forma podemos configurar dinámicamente o en tiempo de compilación y *testing*, si se quiere realmente acceder a los repositorios reales de datos (Bases de datos, etc.), a un segundo sistema de repositorios/almacén diferente o incluso si se quiere acceder a 'falsos repositorios' (repositorios *stub* o *fake repositories*)** de forma que si lo único que queremos hacer es ejecutar un gran volumen de pruebas unitarias siempre justo después de realizar cambios en la lógica de negocio y compilar, esto se realizará de una forma rápida y ágil (sin ralentizar el desarrollo) porque no estaremos accediendo a bases de datos al realizar dichas pruebas unitarias (solo a repositorios de tipo '*mock*' o '*stub*') para realizar dicho gran volumen de pruebas unitarias. Adicionalmente deberemos poder realizar 'pruebas de integración' donde ahí si se realizarán las pruebas contra la Base de Datos real a la que acceden los Repositorios.

Normalmente un método de una clase SERVICIO de Aplicación, invocará a otros objetos (del dominio o de persistencia de datos), formando reglas o transacciones completas **(como en el ejemplo un método que implemente una Transferencia Bancaria**, llamado *BankingManagementService::PerformTransfer()*, que realizaría una llamada al Dominio para que se realicen las operaciones de negocio relativas a la transferencia (internamente en el Dominio, *::Credit()* y *::Debit()*), y posteriormente llamando a los métodos de persistencia de Repositorios para que la transferencia quede grabada/reflejada en el almacén persistente (una base de datos, probablemente). **Todas esas llamadas entre diferentes objetos de las diferentes capas (especialmente en lo relativo a infraestructura) deberán ser llamadas desacopladas mediante interfaces e inyección de dependencias.** El único caso que no tiene mucho sentido desacoplar mediante DI son las entidades del Dominio, pues en las entidades del Dominio no es normal querer sustituirlas por otra versión que cumpla el mismo interfaz.

3.2.1.- Patrón 'Unidad de Trabajo' (UNIT OF WORK)

El concepto del patrón de UNIT OF WORK (UoW) está muy ligado al uso de REPOSITORIOS. En definitiva, un Repositorio no accede directamente a los almacenes (comúnmente bases de datos) de una forma directa cuando le decimos que realice una actualización (*update/insert/delete*). Por el contrario, lo que realiza es un registro en memoria de las operaciones que 'quiere' realizar. Y para que realmente se realicen sobre el almacén o base de datos, es necesario que un ente de mayor nivel aplique dichos cambios a realizar contra el almacén. Dicho ente o concepto de nivel superior es el UNIT OF WORK.

El patrón de UNIT OF WORK encaja perfectamente con las transacciones, pues podemos hacer coincidir un UNIT OF WORK con una transacción, de forma que justo antes del 'commit' de una transacción aplicaríamos con el UoW las diferentes operaciones, agrupadas todas de una vez, con lo que el rendimiento se optimiza y

especialmente se minimizan los bloqueos en base de datos. Por el contrario, si hiciéramos uso solamente de clases de acceso a datos (tradicionales DAL) dentro de una transacción, la transacción tendría una mayor duración y nuestros objetos estarían aplicando operaciones de la transacción mezcladas en el tiempo con lógica del dominio, por lo que el tiempo puramente para la transacción será siempre mayor con el consiguiente aumento de tiempo en bloqueos.

El patrón UNIT OF WORK fué definido por Martin Fowler (Fowler, Patterns of Enterprise Application Architecture, 184). De acuerdo con Martin, "*Un UNIT OF WORK mantiene una lista de objetos afectados por una transacción de negocio y coordina la actualización de cambios y la resolución de problemas de concurrencia*".

El diseño del funcionamiento de un UNIT OF WORK puede realizarse de diferentes formas, pero probablemente el más acertado (como adelantábamos antes) consiste en que los Repositorios deleguen al UNIT OF WORK (UoW) el trabajo de acceder al almacén de datos. Es decir, el UoW será el que realice efectivamente las llamadas al almacén (en bases de datos, comunicar al servidor de base de datos que ejecute sentencias SQL). El mayor beneficio de esta aproximación es que los mensajes que manda el UoW son transparentes al consumidor de los repositorios, puesto que los repositorios solamente le dicen al UoW operaciones que deberá hacer cuando decida aplicar la unidad de trabajo.

El siguiente esquema sería el funcionamiento de las tradicionales/antiguas clases de acceso a datos (DAL), sin utilizar ningún UoW:

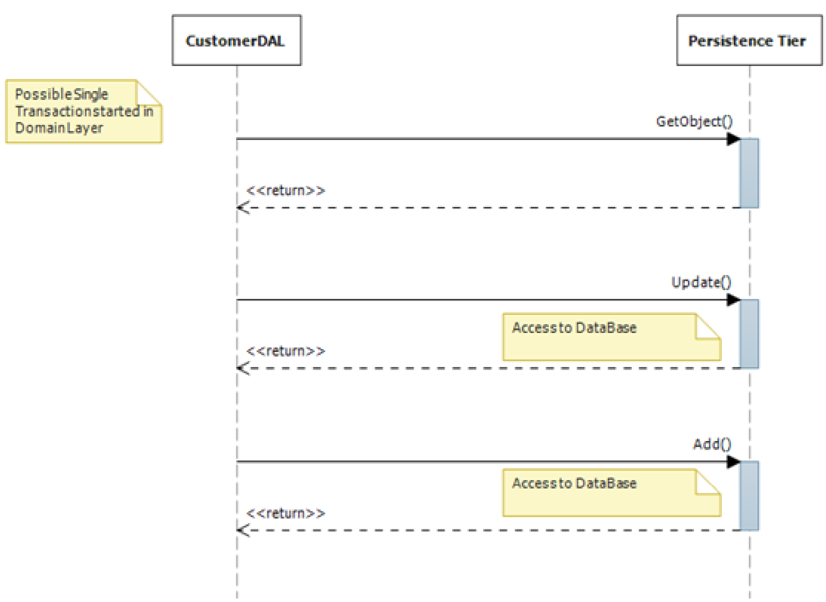


Figura 4.- Esquema clases de acceso a datos (DAL),

El siguiente esquema sería el funcionamiento de una clase REPOSITORY, con un UoW coordinando, que es como recomendamos en esta guía de Arquitectura:

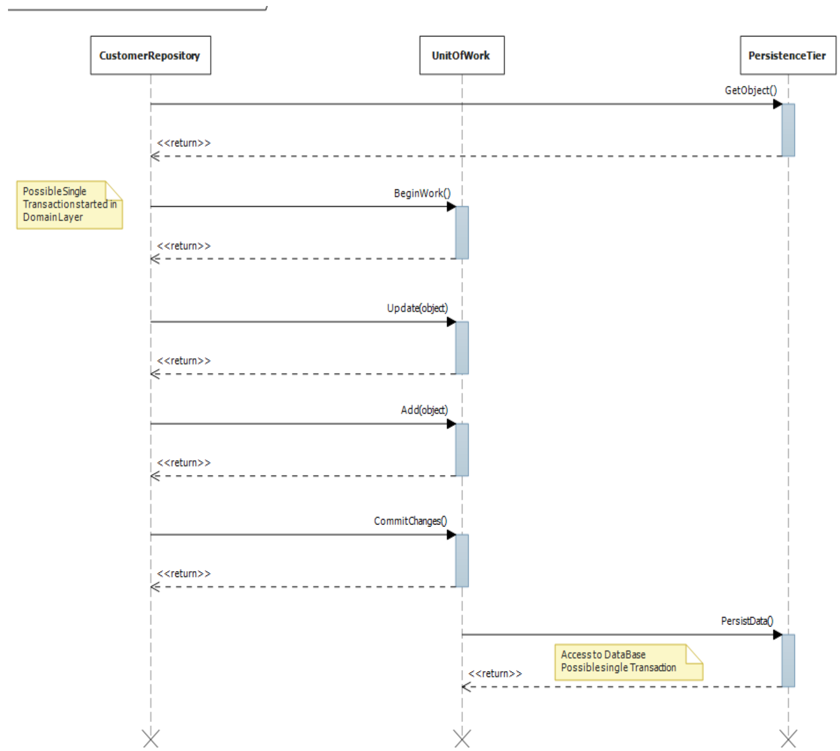


Figura 5.- Funcionamiento de UoW y clase REPOSITORY

Se puede apreciar perfectamente aquí la diferencia entre el funcionamiento de un Repositorio junto con una ‘Unidad de Trabajo’ (UoW) *versus* simples clases de Acceso a Datos (DAL). Haciendo uso de un UoW, las operaciones que realizamos contra los Repositorios, realmente no se realizan hasta que el UoW lo hace, y entonces se aplican todos los cambios ‘registrados’ por hacer de una forma conjunta/seguida (Unidad de Trabajo).

3.2.2.- Servicios Workflows de Capa de Aplicación (Opcional)

Realmente esta sub-capa se trata de un caso especial de SERVICIOS de Aplicación que viene a dar solución a algunas casuísticas determinadas dentro de distintas soluciones de software. Los procesos de larga duración o bien los procesos en los que hay interacción, tanto humana como de otros sistemas software, son un ejemplo claro de uso de flujos de trabajo. Modelar un proceso de interacción humana, por ejemplo, directamente en el lenguaje de codificación seleccionado suele oscurecer demasiado el verdadero propósito del mismo, impidiendo en muchos casos una posible comprensión

del mismo y por lo tanto disminuyendo la legibilidad. Al contrario, una capa de flujo de trabajos nos permite modelar las distintas interacciones por medio de actividades y un diseñador de control que de forma visual no da una idea clara del propósito del proceso a realizar.

Regla Nº: D21. **Diseñar e implementar una sub-capa de servicios de Workflows de Capa de Aplicación**

Recomendaciones

- Esta capa es opcional, no siempre es necesaria, de hecho en aplicaciones muy centradas en datos sin procesos de negocio con interacciones humanas, no es común encontrársela.

- Tratar de encapsular en 'Actividades' los procesos dentro de un flujo de trabajo de tal forma que las mismas sean reutilizables en otros flujos.

- Si bien los flujos de trabajo pueden implementar 'negocio', es recomendable apoyarse siempre en servicios del dominio y repositorios para realizar las distintas tareas que tengan asignadas las actividades del mismo.

Referencias

- **Workflow Patterns** - http://www.workflowpatterns.com/.

Cuando se habla de flujos de trabajo, se habla generalmente de los pasos de estos flujos de trabajo, a los que generalmente se les conoce como actividades. Cuando se hace una implementación de esta subcapa es importante rentabilizar al máximo la reutilización de las mismas y prestar atención a como se implementan, como puntos importantes cabrían destacar los siguientes.

- Si las actividades hacen uso de mecanismos de persistencia deberían utilizar siempre que sea posible los repositorios ya definidos.

- Las actividades pueden orquestar diferentes métodos de la subcapa de aplicación y servicios del dominio.

La mayoría de los motores de flujo de trabajos existentes hoy en dia disponen de mecanismos para garantizar la durabilidad de los mismos ante procesos de larga duración y/o caidas de sistema. De forma usual estos sistemas se basan en bases de datos relaciones y por lo tanto dispondrán de distintas operacionesn que podrían ser susceptibles de incorporar dentro de esta sub capa, algunos ejemplos de operaciones podrían ser las siguientes.

- Rehidratación de los flujos de trabajo desde el sistema de persistencia a la memoria.
- Descarga de los flujos de trabajo desde la memoria hasta el sistema de persistencia.
- Comprobación de la existencia de un determinado flujo de trabajo en el sistema de persistencia.
- Almacenamiento de las correlaciones de las instancias de los flujos de trabajo en el sistema de persistencia.

3.3.- Errores y anti-patrones en la Capa de Aplicación

Hay algunos puntos problemáticos y errores comunes en muchas aplicaciones, que normalmente debemos analizar al diseñar la capa de Aplicación. La siguiente tabla lista dichos puntos, agrupados por categorías.

Tabla 2.- Antipatrones en Capa de Aplicación

Categoría	Errores comunes
Autenticación	- Aplicar autenticación propia de la aplicación, en capas propias de la aplicación, cuando no se requiere y se podría utilizar una autenticación global fuera de la propia aplicación. - Diseñar un mecanismo de autenticación propio - No conseguir un 'Single-Sign-on' cuando sería apropiado
Autorización	- Uso incorrecto de granularidad de roles - Uso de impersonación y delegación cuando no se requiere - Mezcla de código de autorización con código de proceso de negocio

Componentes de Aplicación	- Mezclar en los Servicios de aplicación la lógica de acceso a datos (TSQL, Linq, etc.). - Sobrecarga de los componentes de negocio al mezclar funcionalidad no relacionada. - No considerar el uso de interfaces basados en mensajes (Web-Services) al exponer los componentes de negocio.
Cache	- Hacer cache de datos volátiles - Cachear demasiados datos en las capas de aplicación - No conseguir cachear datos en un formato listo para usar. - Cachear datos sensibles/confidenciales en un formato no cifrado.
Acoplamiento y Cohesión	- Diseño de capas fuertemente acopladas entre ellas. - No existe una separación clara de responsabilidades (*concerns*) entre las diferentes capas.
Concurrencia y Transacciones	- No se ha elegido el modelo correcto de concurrencia de datos - Uso de transacciones ACID demasiado largas que provocan demasiados bloqueos en las bases de datos.
Acceso a Datos	- Acceso a la base de datos directamente desde las capas de negocio/aplicación - Mezcla en los componentes de negocio de lógica de acceso a datos con lógica de negocio.
Gestión de Excepciones	- Mostrar información confidencial al usuario final (como *strings* de conexión al producirse errores) - Uso de excepciones para controlar el flujo de la aplicación - No conseguir mostrar al usuario mensajes de error con información útil.

Instrumentalización y *Logging*	- No conseguir adecuar la instrumentalización en los componentes de negocio - No hacer log de eventos críticos de negocio o eventos críticos del sistema
Validación	- Fiarse exclusivamente de la validación realizada en la capa de presentación - No validar correctamente longitud, rango, formato y tipo - No reusar lógica de validación
Workflow	- No considerar requerimientos de gestión de aplicación - Elegir un patrón de workflow incorrecto. - No considerar como gestionar todas las excepciones de estados. - Elegir una tecnología de workflow incorrecta

3.4.- Aspectos de Diseño relacionados con la Capa de Aplicación

Los siguientes puntos son en su mayoría aspectos transversales de una Arquitectura y se explican en detalle en el capítulo de ***'Capas de Infraestructura Transversal/Horizontal'***, sin embargo es importante reflejar cuales de dichos aspectos están relacionados con la Capa de Aplicación.

3.4.1.- Autenticación

Diseñar una estrategia efectiva de autenticación para la Capa de aplicación, es algo fundamental de cara a la seguridad y fiabilidad de la aplicación. Si esto no se diseña e implementa correctamente, la aplicación puede ser vulnerable a ataques. Se deben considerar las siguientes guías a la hora de definir el tipo de autenticación de la aplicación:

- No realizar la autenticación en la Capa de Aplicación si solo se utilizará por una capa de presentación o un nivel de Servicios-Distribuidos (Servicios-Web, etc.) dentro de la misma frontera de confianza. En estos casos (lo más común en aplicaciones de negocio), la mejor solución es propagar la identidad del cliente a las capas de Aplicación y de Dominio para los casos en los que se debe autorizar basándose en la identidad del cliente inicial.

- Si la Capa de Aplicación y Dominio se utilizarán en múltiples aplicaciones con diferentes almacenes de usuarios, se debe considerar el implementar un sistema de *"single sign-on"*. Evitar diseñar mecanismos de autenticación propios y preferiblemente hacer uso de una plataforma genérica.

- Considerar el uso de "Orientación a *Claims*", especialmente para aplicaciones basadas en Servicios-Web. De esta forma se pueden aprovechar los beneficios de mecanismos de identidad federada e integrar diferentes tipos y tecnologías de autenticación.

Este aspecto transversal (Autenticación) se explica en más detalle en el capítulo ***'Capas de Infraestructura Transversal/Horizontal'***.

3.4.2.- Autorización

Diseñar una estrategia efectiva de autorización para la Capa de aplicación, es algo fundamental de cara a la seguridad y fiabilidad de la aplicación. Si esto no se diseña e implementa correctamente, la aplicación puede ser vulnerable a ataques. Se deben considerar las siguientes guías a la hora de definir el tipo de autorización de la aplicación:

- Proteger recursos de la Capa de Aplicación y Dominio (Clases de servicios, etc.) aplicando la autorización a los consumidores (clientes) basándonos en su identidad, roles, *claims* de tipo rol, u otra información contextual. Si se hace uso de roles, intentar minimizar al máximo el número de roles para poder reducir el número de combinaciones de permisos requeridos.

- Considerar el uso de autorización basada en roles para decisiones de negocio, autorización basada en recursos para auditorías de sistema, y autorización basada en *claims* cuando se necesita soportar autorización federada basada en una mezcla de información como identidad, rol, permisos, derechos y otros factores.

- Evitar el uso de impersonación y delegación siempre que sea posible porque puede afectar de forma significativa al rendimiento y a la escalabilidad. Normalmente es más costoso en rendimiento el impersonar un cliente en una llamada que hacer en sí la propia llamada.

- No mezclar código de autorización.

Este aspecto transversal (Autorización) se explica en más detalle en el capítulo ***'Capas de Infraestructura Transversal/Horizontal'***.

3.4.3.- Cache

Diseñar una estrategia efectiva de *cache* para la aplicación es algo fundamental de cara al rendimiento y escalabilidad de la aplicación. Se debe hacer uso de Cache para optimizar consultas de datos maestros, evitar llamadas remotas innecesarias por red y en definitiva eliminar procesos y consultas duplicadas. Como parte de la estrategia se debe decidir cuándo y cómo cargar datos en la cache. Es algo completamente dependiente de la naturaleza de la Aplicación y del Dominio, pues depende de cada entidad.

Para evitar esperas innecesarias del cliente, cargar los datos de forma asíncrona o hacer uso de procesos *batch*.

Se deben considerar las siguientes guías a la hora de definir la estrategia de cache de la aplicación:

- Hacer cache de datos estáticos que se reutilizarán regularmente en las diferentes capas (finalmente se utilizarán/manejarán en la Capa de Dominio y en Presentación), pero evitar hacer cache de datos muy volátiles. Considerar hacer cache de datos que no pueden ser obtenidos de la base de datos de forma rápida y eficiente pero evitar hacer cache de volúmenes muy grandes de datos que pueden ralentizar el proceso. Hacer cache de volúmenes mínimos requeridos.

- Evitar hacer cache de datos confidenciales o bien diseñar un mecanismo de protección de dichos datos en la cache (como cifrado de dichos datos confidenciales).

- Tener en cuenta despliegues en 'Granjas de Servidores Web', lo cual puede afectar a caches estándar en el espacio de memoria de los Servicios. Si cualquiera de los servidores del Web-Farm puede gestionar peticiones del mismo cliente (Balanceo sin afinidad), la cache a implementar debe soportar sincronización de datos entre los diferentes servidores del Web-*Farm*. Microsoft dispone de tecnologías adecuadas a este fin (Caché Distribuido), como se explica más adelante en la guía.

Este aspecto transversal (Cache) se explica en más detalle en el capítulo ***'Capas de Infraestructura Transversal/Horizontal'***.

3.4.4.- Gestión de Excepciones

Diseñar una estrategia efectiva de *Gestión de Excepciones* para la Capa de aplicación, es algo fundamental de cara a la estabilidad e incluso a la seguridad de la aplicación. Si no se realiza una gestión de excepciones correcta, la aplicación puede ser vulnerable a ataques, puede revelar información confidencial de la aplicación, etc. Así mismo, el originar excepciones de negocio y la propia gestión de excepciones son operaciones con un coste de proceso relativamente 'caro' en el tiempo, por lo que es importante que el diseño tenga en cuenta el impacto en el rendimiento.

Al diseñar la estrategia de gestión de excepciones, deben considerarse las siguientes guías:

- Capturar (*Catch*) solamente las excepciones que se puedan realmente gestionar o si se necesita añadir información.

- Bajo ningún concepto se debe hacer uso del sistema de control de excepciones para controlar el flujo de aplicación o lógica de negocio, porque la implementación de capturas de excepciones (*Catch*, etc.) tiene un rendimiento muy bajo y en dichos casos (puntos de ejecución normales de la aplicación) impactaría muy desfavorablemente en el rendimiento de la aplicación.

- Diseñar una estrategia apropiada de gestión de excepciones, por ejemplo, permitir que las excepciones fluyan hasta las capas 'frontera' (último nivel del servidor de componentes, por ejemplo) y será ahí donde pueden/deben ser persistidas en un sistema de '*logging*' y/o transformadas según sea necesario antes de pasarlo a la capa de presentación. Es bueno también incluir un identificador de contexto de forma que las excepciones relacionadas puedan asociarse a lo largo de diferentes capas y se pueda fácilmente identificar el origen/causa de los errores.

Este aspecto transversal (Gestión de Excepciones) se explica en más detalle en el capítulo ***'Capas de Infraestructura Transversal/Horizontal'***.

3.4.5.- Logging, Auditoría e Instrumentalización

Diseñar una estrategia efectiva de *Logging, Auditoría e Instrumentalización* para la Capa de Dominio y Aplicación es importante para la seguridad, estabilidad y mantenimiento de la aplicación. Si no se diseña e implementa correctamente, la aplicación puede ser vulnerable a acciones de repudio cuando ciertos usuarios nieguen

sus acciones. Los ficheros de log/registro pueden ser requeridos para probar acciones incorrectas en procedimientos legales. La Auditoría se considera más precisa si el log de información se genera en el preciso momento del acceso al recurso y por la propia rutina que accede al recurso.

La instrumentalización puede implementarse con eventos y contadores de rendimiento, así como utilizar posteriormente herramientas de monitorización para proporcionar a los administradores información sobre el estado, rendimiento y salud de la aplicación.

Considerar las siguientes guías:

- Centralizar el *logging*, auditorías e instrumentalización en las capas de Aplicación y Dominio.

- Se puede hacer uso de clases/librerías sencillas reutilizables, o para aspectos más avanzados (publicación transparente en diferentes repositorios e incluso traps SNMP) se recomienda hacer uso de librerías como '***Microsoft Patterns & Practices Enterprise Library***' o de terceras partes como *Apache Logging Services* "*log4Net*" o Jarosław Kowalski's "*NLog*".

- Incluir instrumentalización en eventos críticos del sistema y/o de negocio dentro de los componentes de la Capa de Aplicación y Capa de Dominio

- No almacenar información confidencial en los ficheros de Log

- Asegurarse de que los fallos en el sistema de *logging* no afectan al funcionamiento normal de la capa de Aplicación y Dominio.

3.4.6.- Validaciones

Diseñar una estrategia efectiva de validaciones en la Capa de Aplicación y Dominio es importante para la estabilidad de la aplicación, pero también para la usabilidad de la aplicación hacia el usuario final. Si no se realiza apropiadamente puede dar lugar a inconsistencias de datos y violaciones de reglas de negocio, y finalmente una experiencia de usuario muy mediocre debido a errores originados posteriormente que se podrían haber detectado mucho antes. Además, si no se realiza correctamente, la aplicación puede ser también vulnerable a aspectos de seguridad como ataques '*Cross-Site-Scripting*' en aplicaciones web, ataques de inyecciones SQL, '*buffer overflow*', etc.

Considerar las siguientes guías:

- Validar todos los datos de entrada y parámetros de métodos en la capa de Aplicación, incluso aunque se haya realizado una validación de datos anterior en la capa de presentación. La validación de datos en la capa de presentación

está más relacionada con la experiencia de usuario y la realizada en la Capa de Apliación está más relacionada con aspectos de seguridad de la aplicación.

- Centralizar la estrategia de validación para facilitar las pruebas y la reutilización.

- Asumir que todos los datos de entrada de un usuario pueden ser 'maliciosos'. Validar longitud de datos, rangos, formatos y tipos así como otros conceptos más avanzados del negocio/dominio.

3.4.7.- Aspectos de despliegue de la Capa de Aplicación

Al desplegar la capa de Aplicación y Dominio, tener en cuenta aspectos de rendimiento y seguridad del entorno de producción. Considerar las siguientes guías:

- Considerar un despliegue de la capa de aplicación y dominio en mismo nivel físico que el nivel de presentación web si se quiere maximizar el rendimiento. Solo se debe separar a otro nivel físico por aspectos de seguridad y de algunos casos especiales de escalabilidad.

3.4.8.- Concurrencia y Transacciones

Cuando se diseña para aspectos de Concurrencia y Transacciones, es importante identificar el modelo apropiado de concurrencia y determinar cómo se gestionarán las transacciones. Para la concurrencia se puede escoger entre el modelo optimista y el pesimista.

Modelo de Concurrencia Optimista

En este modelo, los bloqueos no se mantienen en la base de datos (solo el mínimo imprescindible mientras se actualiza, pero no mientras el usuario está trabajando o simplemente con la ventana de actualización abierta) y por lo tanto las actualizaciones requieren el realizar comprobaciones de que los datos no han sido modificados en la base de datos desde la obtención original de los datos a modificar. Normalmente se articula en base a *timestamps* (sello de tiempo).

Modelo de Concurrencia Pesimista

Los datos a actualizar se bloquean en la base de datos y no pueden ser actualizados por otras operaciones hasta que se hayan desbloqueado.

Considera las siguientes guías relativas a concurrencia y transacciones:

- Se deben tener en cuenta las fronteras de la transacción de forma que se puedan realizar reintentos y composiciones.

- Cuando no se pueda aplicar un '*commit*' o un '*rollback*' o si se hace uso de transacciones de larga ejecución, elegir mejor la opción de implementar métodos compensatorios para deshacer las operaciones realizadas sobre los datos y dejarlos en su estado anterior en caso de que una operación falle. Esto es debido a que no se puede mantener bloqueada una base de datos debido a una transacción de larga duración.

- Evitar mantener bloqueos durante largos períodos de tiempo, por ejemplo no realizar transacciones de larga duración que sean '*Two Phase Commit*'.

- Elegir un nivel apropiado de aislamiento de la transacción. Este nivel define como y cuando los cambios estarán disponibles a otras operaciones.

3.5.- Mapa de patrones posibles a implementar en la capa de Aplicación

En la siguiente tabla de muestran los patrones clave para las capas de aplicación, organizados por categorías. Es importante considerar el uso de dichos patrones cuando se toman las decisiones para cada categoría.

Tabla 3.- Patrones Clave

Categorías	Patrones
Componentes de Capa de Aplicación	• Application Façade • Chain of Responsibility • Command
Concurrencia y Transacciones	• Capture Transaction Details • Coarse-Grained Lock • Implicit Lock • Optimistic Offline Lock

	• Pessimistic Offline Lock • Transaction Script
Workflows	• Data-driven workflow • Human workflow • Sequential workflow • State-driven workflow

Referencias de patrones

Información sobre patrones ‘Command’, ‘Chain of Responsability’ y ‘Façade’
o “data & object factory” at **http://www.dofactory.com/Patterns/Patterns.aspx**

Información sobre patrón “Entity Translator”
o **http://msdn.microsoft.com/en-us/library/cc304800.aspx**

Patrón “Capture Transaction Details pattern”, ver “Data Patterns” en **http://msdn.microsoft.com/en-us/library/ms998446.aspx**

4.- IMPLEMENTACIÓN EN .NET DE CAPA DE APLICACION

La explicación y definición lógica de esta capa está explicada en la sección anterior, por lo que en la presente sección nos centramos en mostrar las posibilidades de implementación de la Capa de Aplicación, en .NET 4.0.

En el siguiente diagrama resaltamos la situación de la Capa de Aplicación, pero en este caso haciendo uso ya de un *Diagrama Layer* implementado con Visual Studio 2010 y con un mapeo real de cada capa a los diferentes *namespaces* que las implementan:

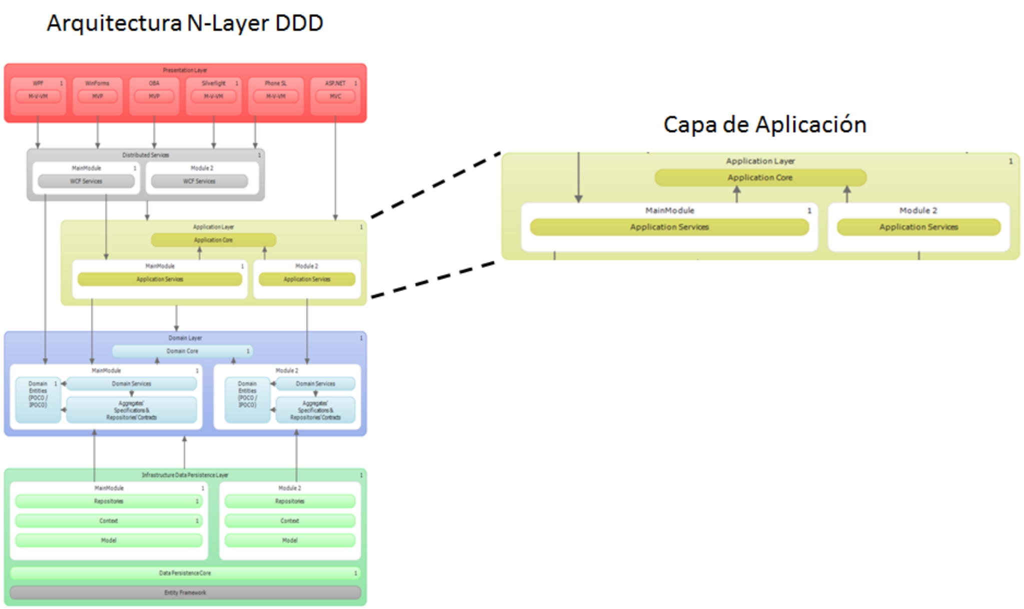


Figura 6.- Diagrama situación Capa de Aplicación

Pasos a realizar:

4.- Una vez identificadas las áreas de la aplicación que son características y los requerimientos del software, no del Dominio, entonces debemos crear la estructura de esta capa, es decir, el o los proyectos en Visual Studio que alojarán las clases .NET implementando los SERVICIOS de Aplicación.

5.- Iremos añadiendo e implementando clases .NET de SERVICIOS de Aplicación según necesitemos. Es importante recordar que en esta capa también debemos seguir trabajando con abstracciones (Interfaces). Así pues, por cada clase de implementación de un SERVICIO, deberemos disponer

también de un interfaz con la declaración de sus operaciones respectivas. Este interfaz será utilizado desde la capa superior (Servicios Web o Presentación en ASP.NET) con el contenedor Unity, pidiéndole al contenedor de UNITY que resuelva un objeto para el interfaz de Servicio que le pedimos. El proceso es similar al seguido en la implementación de SERVICIOS del Dominio. Lo que cambia en este caso es el contenido de los SERVICIOS, en lugar de la lógica del Dominio (lógica de negocio), en este caso implementaremos lógica de coordinación de tareas requeridas por el software en sí (coordinación de persistencia, integraciones, optimizaciones, etc.)

6.- Cabe la posibilidad de que la implementación de los SERVICIOS de la capa de aplicación se implementen con tecnologías de WORKFLOW, no solamente mediante clases .NET como única posibilidad.

4.1.- Implementación de Servicios de Capa de Aplicación

Los SERVICIOS de APLICACIÓN deben ser, normalmente y salvo pocas excepciones, el único punto o tipo de componente de la arquitectura por el que se acceda a las clases de infraestructura de persistencia de datos (*Repositorios*). No se debe de acceder directamente a los Repositorios desde Capas de Presentación o Servicios-Web. En caso contrario, nos estaríamos saltando lógica de aplicación y también la de negocio/dominio.

En el gráfico siguiente, podemos ver las clases *Servicio* (de Aplicación) y las clases *Repositorios* relacionadas (Los Repositorios forman parte de la capa de Infraestructura de Persistencia de Datos), de un módulo ejemplo de aplicación:

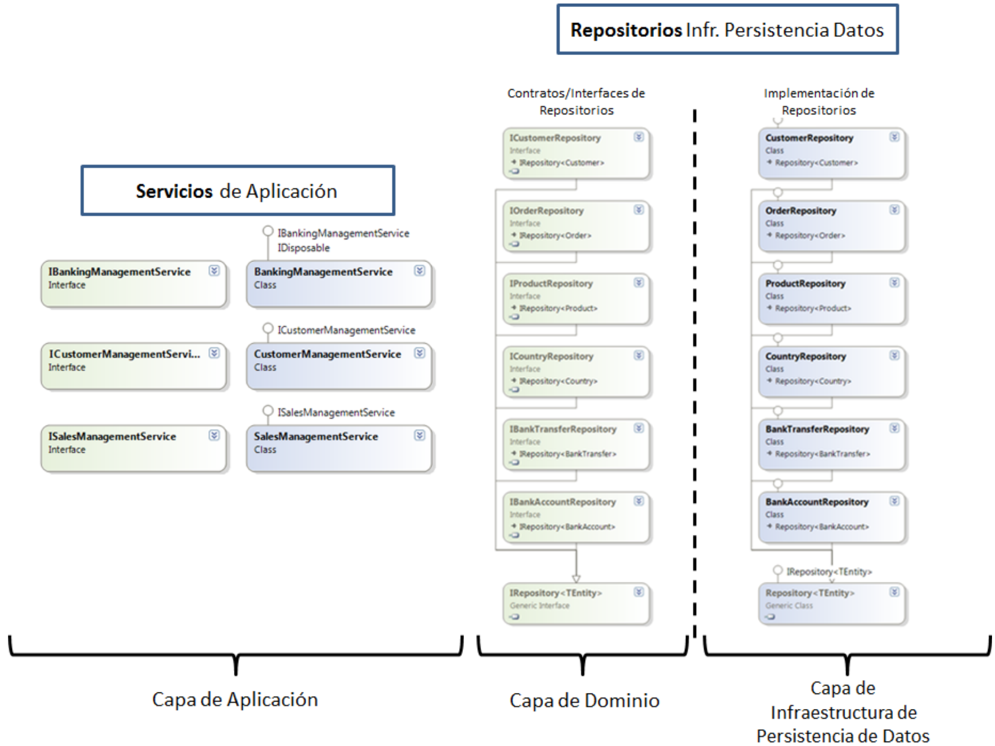


Figura 7.- Gráfico Clases de Servicios de Aplicación y Repositorios

Adicionalmente a las clases de Servicios de Aplicación y Repositorios, también tenemos los Servicios del Dominio. Sin embargo en el diagrama anterior los obviamos porque la relación con los Repositorios (creación y uso de Repositorios) la haremos normalmente de forma mayoritaria desde los Servicios de la Capa de Aplicación.

A continuación se muestra un ejemplo de implementación de clase de SERVICIO de Aplicación para controlar lo relativo a la entidad **Customer**:

C#

Interfaz para abstracción e instanciación mediante contenedor IoC (Unity), desde capas superiores (p.e. Web-Services)

```
public class CustomerManagementService : ICustomerManagementService
{
    ICustomerRepository _CustomerRepository;
```

Constructor con Dependencia requerida (Repositorio) a ser inferido e instanciado por el contenedor IoC (Unity).

```
    public CustomerManagementService(ICustomerRepository
customerRepository)
    {
        _CustomerRepository = customerRepository;
    }
```

```
                    Lógica de Aplicación para la entidad 'Customer'.

    public List<Customer> FindPagedCustomers(int pageIndex, int
pageCount)
    {
                    Validaciones y Generación de Excepciones de Negocio

        if (pageIndex < 0)
                throw new
ArgumentException(Resources.Messages.exception_InvalidPageIndex,
"pageIndex");

        if (pageCount <= 0)
                throw new
ArgumentException(Resources.Messages.exception_InvalidPageCount,
"pageCount");


        Specification<Customer> onlyEnabledSpec = new
DirectSpecification<Customer>();

                    Acceso a Fuentes de Datos mediante Repositorios.

        return _customerRepository.GetPagedElements(pageIndex,
pageCount, c => c.CustomerCode, onlyEnabledSpec, true)
                                      .ToList();
    }

    // Otros métodos de CustomerManagementService a implementar
posteriormente (Con patrones UoW y Specifications)
    // ...

}
```

Todo el código anterior es bastante claro, excepto probablemente un punto: **¿Dónde se está instanciando y creando el objeto de Repositorio del contrato 'ICustomerRepository'?**

Esto tiene que ver precisamente con la Inyección de Dependencias y el desacoplamiento entre objetos mediante el contenedor IoC de Unity que explicamos a continuación.

4.1.1.- Desacoplamiento e Inyección de Dependencias entre Servicios de Aplicación y Repositorios mediante IoC de UNITY

Al desacoplar los Servicios de la capa de aplicación con respecto a los objetos inferiores como los Repositorios (Pertenecientes a la Capa de Infraestructura de Persistencia de Datos), podemos configurar dinámicamente o en tiempo de

compilación y *testing*, si se quiere realmente acceder a los repositorios reales de datos (Bases de datos, etc.) o a otras implementaciones diferentes de Repositorios con accesos a almacenes de otra naturaleza, o se quiere acceder a 'falsos repositorios' (repositorios *stub* o *fake repositories*) de forma que si lo único que queremos hacer es ejecutar un gran volumen de pruebas unitarias siempre justo después de realizar cambios en la lógica de negocio y compilar, esto se realizará de una forma rápida y ágil (sin ralentizar el desarrollo) porque no estaremos accediendo a bases de datos al realizar dichas pruebas unitarias (solo a repositorios de tipo '*mock*' o '*stub*') para realizar dicho gran volumen de pruebas unitarias. Adicionalmente deberemos poder realizar 'pruebas de integración' donde ahí si se realizarán las pruebas contra la Base de Datos real a la que acceden los Repositorios.

En el siguiente esquema podemos distinguir, en este caso, donde se está implementando Inyección de dependencias con UNITY, entre las clases de 'Servicios de la Aplicación' y los Repositorios de la capa de 'Infraestructura de Persistencia y Acceso a Datos':

Inyección de dependencias de Repositorios en los Servicios de Aplicación

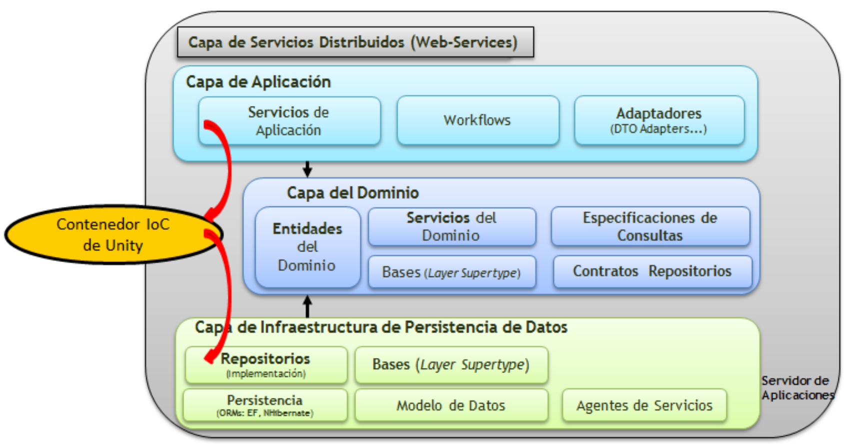


Figura 8.- Esquema Servicios de Dominio

A continuación vamos a ver cómo se puede realizar dicha integración desacoplada entre ambas capas (componentes del dominio y Repositorios), pero si no se conoce Unity, es importante leer primero el capítulo "Implementación de Inyección de Dependencias e IoC con UNITY" que forma parte de esta guía de Arquitectura e implementación.

Registro de clases e interfaces en el contenedor de Unity

Antes de poder instanciar ninguna clase a través del contenedor de Unity, lógicamente, necesitamos 'registrar' los tipos en el contenedor IoC de Unity, tanto los interfaces como las clases. **Este proceso de registro se puede hacer por código**

compilado (C#, etc.) o también de forma declarativa mediante el XML de configuración de Unity.

En el caso de registrar los tipos de clases y los mapeos utilizando XML, entonces se puede optar por mezclar el XML de configuración de Unity con el XML del web.config o App.config del proceso que hospede nuestra aplicación/servicio, o mejor aún (más limpio), también podemos disponer de un fichero XML específico para Unity enlazado a nuestro fichero de configuración app.config/web.config. En la implementación ejemplo estamos utilizando un fichero de configuración específico para Unity, llamado Unity.config.

Este sería el XML de enlace desde el web/app .config al fichero de configuración de Unity:

```
Web.config (De Servicio WCF, o app ASP.NET, etc.)
…
  <!-- Unity configuration for solving dependencies-->
  <unity configSource="Unity.config"/>
…
```

Este es el XML de configuración para registrar el interfaz y clase del Repositorio:

```
Web.config (De Servicio WCF, etc.)
…
  <!-- Unity configuration for solving dependencies-->
  <unity configSource="Unity.config"/>
…
XML – Unity.config
<?xml version="1.0" encoding="utf-8" ?>
<unity>
  <typeAliases>
   …
   …
```

Registro de Contrato/Interfaz del Repositorio.

```
    <typeAlias alias="ICustomerRepository"

type="Microsoft.Samples.NLayerApp.Domain.MainModule.Contracts.ICustomerR
epository,
            Microsoft.Samples.NLayerApp.Domain.MainModule" />
```

Registro de la clase del Repositorio.

```
    <typeAlias alias="CustomerRepository"

type="Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule.Reposit
ories.CustomerRepository,
            Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule"
/>

…
…
```

A continuación viene la parte interesante, es decir, el mapeo que podemos especificarle al contenedor entre los contratos/interfaces y la clase que debe de instanciar el contenedor de Unity. Es decir, un mapeo que diga "*Cuando pida un objeto*

para ICustomerRepository, instancia y dame un objeto de la clase CustomerRepository". Pero lo interesante es que en otro momento podría especificar algo similar a lo siguiente si quiero hacer pruebas unitarias contra una implementación falsa, un stub/mock: "*Cuando pida un objeto para ICustomerRepository, instancia un objeto de la clase CustomerFakeRepository*".

Así pues, el XML declarativo en el fichero Unity.config donde especificamos dicho mapeo para nuestro Repositorio ejemplo, es el siguiente:

XML – Unity.config

```
<?xml version="1.0" encoding="utf-8" ?>
<unity>
  <typeAliases>
   …
   …
  </typeAliases>
  …
  …

<!-- UNITY CONTAINERS -->
  <containers>
    <container name="RootContainer">
      <types>
        …
        <type type="ICustomerRepository" mapTo="CustomerRepository">
        </type>
        …
      </types>
    </container>

…
…
```

Contenedor. Podemos tener una jerarquía de contenedores, creada por programa. Aquí solo podemos definir los mapeos de cada contenedor.

Mapeo de Interfaz a Clase que será instanciada por el contenedor de Unity

Este registro de tipos y mapeo de interfaces a clases, también podemos realizarlo mediante código .NET (C#, VB, etc.), que probablemente es más cómodo mientras se está en pleno desarrollo del proyecto. Con código C# es como está hecho en la aplicación ejemplo, con un código similar al siguiente, en la clase *factory* de IoC:

```
            //Register Repositories mappings
            container.RegisterType<IProductRepository,
ProductRepository>(new TransientLifetimeManager());
            container.RegisterType<IOrderRepository,
OrderRepository>(new TransientLifetimeManager());
            container.RegisterType<IBankAccountRepository,
BankAccountRepository>(new TransientLifetimeManager());
            container.RegisterType<ICustomerRepository,
CustomerRepository>(new TransientLifetimeManager());
            container.RegisterType<ICountryRepository,
CountryRepository>(new TransientLifetimeManager());

            //Register application services mappings

            container.RegisterType<ISalesManagementService,
SalesManagementService>(new TransientLifetimeManager());
            container.RegisterType<ICustomerManagementService,
```

```
CustomerManagementService>(new TransientLifetimeManager());
            container.RegisterType<IBankingManagementService,
BankingManagementService>(new TransientLifetimeManager());

            //Register domain services mappings
            container.RegisterType<IBankTransferDomainService,
BankTransferDomainService>(new TransientLifetimeManager());


            //Register crosscuting mappings
            container.RegisterType<ITraceManager, TraceManager>(new
TransientLifetimeManager());
   …
   …
```

Una vez tenemos definidos los mapeos, podemos proceder a implementar el código donde realmente se pide al contenedor de Unity que nos instancie un objeto para un interfaz dado. Podríamos hacer algo así desde código (Cuidado, que normalmente no haremos un Resolve<> explícito para los Repositorios):

```
   C#

IUnityContainer container = new UnityContainer();
ICustomerRepository customerRep = container.Resolve<ICustomerRepository
>();
```

Es importante destacar que si se quiere aplicar correctamente la DI (Inyección de Dependencias), normalmente haremos un Resolve<> solamente contra las clases de más alto nivel de nuestro servidor de aplicaciones, es decir, desde los puntos entrantes o iniciales, que normalmente son los Servicios-Web (WCF) y/o Capa de Presentación ASP.NET. No deberíamos hacer un Resolve<> explícito contra Repositorios, si no, estaríamos utilizando el *container* casi solamente como selector de tipos. No sería correcto desde el punto de vista de DI.

En definitiva, como debemos tener una cadena de capas integradas con desacoplamiento entre ellas mediante Unity, lo más óptimo es dejar que Unity detecte nuestras dependencias a través de nuestro constructor de cada clase. Es decir, **si nuestra clase de servicio de aplicación tiene una dependencia con una clase de Repositorio (necesitará utilizar un objeto Repositorio), simplemente lo especificamos en nuestro constructor y será el contenedor Unity quien cree el objeto de esa dependencia (el objeto Repositorio) y nos lo proporciona como parámetro de nuestro constructor**.

Así, por ejemplo, nuestra clase de SERVICIO llamada 'CustomerManagement-Service', será así:

```
C#

public class CustomerManagementService : ICustomerManagementService
{
    ICustomerRepository _CustomerRepository;
```

```
    public CustomerManagementService(ICustomerRepository
customerRepository)
    {

        _CustomerRepository = customerRepository;
    }
   …
}
```

Constructor con Dependencia requerida (Repositorio) a ser inferido e instanciado por el contenedor IoC (Unity).

Es importante destacar que, como se puede observar, no hemos hecho ningún ‘new’ explícito de la clase CustomerRepository. Es el contenedor de Unity el que automáticamente crea el objeto de CustomerRepository y nos lo proporciona como parámetro de entrada a nuestro constructor. **Esa es precisamente la inyección de dependencias en el constructor.**

Después, dentro del constructor estamos precisamente guardando la dependencia (Repositorio, en este caso), como variable/objeto miembro, para poder utilizarlo desde los diferentes métodos de nuestro Servicio de la capa de Aplicación.

Así, nuestra clase de servicio de aplicación llamada **CustomerManagementService** quedaría, de forma casi completa, como sigue:

Interfaz para abstracción e instanciación mediante contenedor IoC (Unity)

```
C#

public class CustomerManagementService: ICustomerManagementService
{

   ICustomerRepository _CustomerRepository;
```

Constructor con Dependencia requerida (Repositorio) a ser inferido e instanciado por el contenedor IoC (Unity).

```
   public CustomerManagementService(ICustomerRepository
   customerRepository)
   {
      _CustomerRepository = customerRepository;
   }
```

Lógica de Dominio/Negocio para entidad ‘Customer’.

```
   public List<Customer> FindPagedCustomers(int pageIndex, int
   pageCount)
   {
      if (pageIndex < 0)
                   throw new
ArgumentException(Resources.Messages.exception_InvalidPageIndex,
"pageIndex");

      if (pageCount <= 0)
                   throw new
ArgumentException(Resources.Messages.exception_InvalidPageCount,
```

Validaciones y Generación de Excepciones de Negocio

```
"pageCount");


        return _CustomerRepository.GetPagedElements(pageIndex, pageCount,
c => c.ContactTitle, true).ToList();
    }
                                    Acceso a Fuentes de Datos mediante Repositorios.

    public Customer FindCustomerByCode(string customerCode)
    {
        //Create specification
        CustomerCodeSpecification spec = new
    CustomerCodeSpecification(customerCode);    Uso de Patrón SPECIFICATION

        return _CustomerRepository.FindCustomer(spec);
    }

    public void ChangeCustomer(Customer customer)
    {
                                    Uso de patrón UoW (UNIT OF WORK)
        //Begin unit of work
        IUnitOfWork unitOfWork = _CustomerRepository.StoreContext as
    IUnitOfWork;
        _CustomerRepository.Modify(customer);
        //Complete changes in this unit of work
        unitOfWork.Commit(CommitOption.ClientWins);
    }
}
```

Finalmente y aunque el código que exponemos a continuación no forma parte de esta capa de Aplicación, así es como comenzaría la cadena de creaciones de objetos con inyección de dependencias por constructor. Este código expuesto a continuación se implementaría en una Capa de Servicios WCF o incluso en una capa de presentación web ASP.NET ejecutándose en el mismo servidor de aplicaciones:

```
C# (En Capa de Servicio WCF o en aplicación ASP.NET)
…
{
     IUnityContainer container = new UnityContainer;
     ICustomerService custService =
container.Resolve<ICustomerManagementService>();
     custService.AddCustomer(customer);
}
```

Aunque en la aplicación ejemplo estamos utilizando una clase utilidad estática para Unity (IoCFactory), y el código queda más limpio y extensible:

```
    C# (En Capa de Servicio WCF o en aplicación ASP.NET)
    …
    {
         ICustomerManagementService custService =
ServiceFactory.Current.Resolve<ICustomerManagementService>();
         custService.AddCustomer(customer);
    }
```

El diagrama de clases de servicios de aplicación y Repositorio, solo para lo relativo a la entidad del Dominio “Customer”, quedaría así:

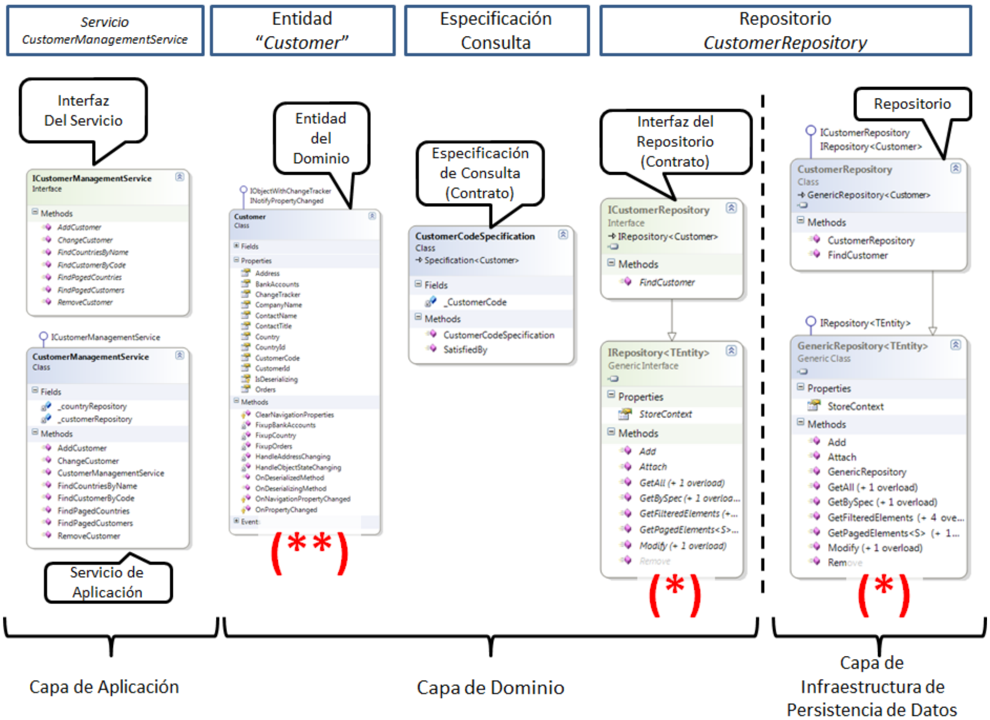


Figura 9.- Diagrama de clases de Servicios de App y Repositorio

Aunque puede parecer que necesitamos muchas clases e interfaces relacionadas con una única entidad del Dominio, son necesarias si se quieren disponer de un desacoplamiento y realmente requiere muy poco trabajo implementarlo, porque:

- De todas estas clases, las marcadas con un (*), en la parte inferior, son clases base, por lo que solo se implementan una única vez para todo el proyecto.

- La clase entidad del Dominio "Customer", marcada con dos asteriscos (**), es generada por el T4 de Entity Framework, por lo que no requiere ningún trabajo.

- Los interfaces son solo declaraciones de métodos, muy rápidos de crear y modificar.

Así pues, solamente necesitamos implementar la propia clase del Servicio "*CustomerManagementService*", con la lógica de capa de Aplicación que requiramos, y también el Repositorio "CustomerRepository" con lógica de persistencia y acceso a datos si no nos resulta reutilizable la que ya tiene la clase base de repositorios.

4.2.- Implementación de Transacciones y UoW en Servicios de Capa de Aplicación

Antes de mostrar la implementación interna del Servicio ejemplo, precisamente porque dicha implementación ejemplo está relacionada con la implementación de transacciones, vamos a mostrar primero las diferentes opciones de implementación de transacciones en .NET y posteriormente lo implementaremos en el código del Servicio ejemplo “BankTransferService”.

4.2.1.- Transacciones en .NET

Una transacción es un intercambio de información y acciones secuenciales asociadas que se tratan como una unidad atómica de forma que se satisfaga una petición y se asegure simultáneamente una integridad de datos concreta. Una transacción solo se considera completa si toda la información y acciones de dicha transacción han finalizado y todos los cambios asociados a bases de datos están aplicados de forma permanente. Las transacciones soportan la acción ‘deshacer’ (*rollback*) cuando se produce algún error, lo cual ayuda a preservar la integridad de datos en las bases de datos.

En .NET se hemos tenido históricamente varias formas posibles de implementar transacciones.

Básicamente, las siguientes opciones:

- Transacciones en TSQL (En las propias sentencias SQL)
- Transacciones ADO.NET (Basadas en los objetos Connection y Transaction)
- Transacciones Enterprise Services (Transacciones distribuidas y basadas en COM+)
- **Transacciones System.Transaction (Locales y promocionables a distribuidas)**

El primer tipo (transacciones en sentencias SQL y/o procedimientos almacenados) es factible para cualquier lenguaje y plataforma de programación (.NET, VB, Java, etc.) y es la que mejor rendimiento puede llegar a tener y para casos concretos y especiales puede ser la más idónea. Sin embargo, no se recomienda hacer uso de ella normalmente en una aplicación de negocio con arquitectura N-Layer, porque tiene el gran inconveniente de tener completamente acoplado el concepto de transacción (es un concepto de negocio, por ejemplo una transferencia) con el código de acceso a datos (sentencias SQL). Recuérdese que una de las normas básicas de una aplicación *N-Layer* es que el código de aplicación y dominio/negocio debe de estar completamente separado y desacoplado del código de persistencia y acceso a datos. **Las transacciones**

deberían declararse/implementarse exclusivamente en la Capa de Aplicación (o Capa de Dominio, dependiendo de preferencias).

Por otro lado, en .NET 1.x, teníamos básicamente dos opciones principales, Transacciones ADO.NET y transacciones COM+ con *Enterprise Services*. Si en una aplicación se utilizaban transacciones ADO.NET, debemos tener en cuenta que estas transacciones están muy ligadas al objeto *Database Connection y Transaction*, que están relacionados con el nivel de acceso a datos y por lo tanto resulta muy difícil poder definir las transacciones exclusivamente en el nivel de componentes de negocio (solamente mediante un Framework propio basado en *aspectos*, etc.). En definitiva, tenemos un problema parecido a utilizar transacciones en sentencias SQL, pero ahora en lugar de definir las transacciones en el propio SQL, estaríamos muy ligados a la implementación de objetos de ADO.NET. Tampoco es el contexto ideal para las transacciones que deberían poder definirse exclusivamente a nivel de negocio.

Otra opción que nos permitía .NET Framework 1.x es utilizar transacciones de *Enterprise Services* (basadas en *COM+*), las cuales sí que se pueden especificar exclusivamente a nivel de clases de negocio (mediante atributos .NET), sin embargo, en este caso tenemos el inconveniente de que su uso impacta gravemente en el rendimiento (*Enterprise Services* se basa en *COM+* y por lo tanto desde .Net se utiliza *COMInterop* y también una comunicación interproceso con el *DTC*), además de que el desarrollo se vuelve algo más tedioso pues se deben firmar los componentes con un nombre seguro (*strong-name*) y registrarlos como componentes *COM* en *COM+*.

Sin embargo, a partir de **.NET 2.0** (continuado en .NET 3.0, 3.5 y 4.0) tenemos el *namespace* '***System.Transactions***'. Esta es, en general, la forma más recomendable de implementar transacciones, por su flexibilidad, mayor rendimiento frente a Enterprise Services especialmente a partir de SQL Server 2005 y su **posibilidad de 'promoción automática de transacción local a transacción distribuida'**.

A continuación se muestra una tabla que sintetiza las diferentes opciones tecnológicas para coordinar transacciones en .NET:

Tabla 4.- Opciones tecnológicas para coordinar transacciones en .NET

Tipo de transacciones	V. Framework .NET	Descripción
Transacciones internas con T-SQL (en BD)	Desde .NET Framework 1.0, 1.1	Transacciones implementadas internamente en las propias sentencias de lenguaje SQL (internamente en procedimientos almacenados, por ejemplo).
Transacciones Enterprise Services (COM+)	Desde .NET Framework 1.0, 1.1	- Enterprise Services (COM+) - Transacciones Web ASP.NET - Transacciones XML Web Services(WebMethod)

Transacciones ADO.NET	Desde .NET Framework 1.0, 1.1	Implementadas con los objetos ADO.NET Transaction y ADO.NET Connection
Transacciones System.Transactions	.NET Framework 2.0, 3.0, 3.5 y 4.0	Potente sistema promocionable de transacciones locales a transacciones distribuidas

Esta otra tabla muestra premisas de recursos y objetivos y qué tecnología de transacciones deberíamos utilizar:

Tabla 5.- Premisas de recursos y objetivos

¿Qué tengo? y Objetivos	¿Qué usar?
- Un Servidor SQL Server 2005/2008/2008R2 para la mayoría de transacciones y también pudieran existir transacciones distribuidas con otros SGBD y/o entornos transaccionales 'Two Phase Commit' - Objetivo: Máximo rendimiento en transacciones locales	→ System.Transactions (A partir de .NET 2.0)
- Un único Servidor SGBD antiguo (Tipo SQL Server 2000), para las mismas transacciones - Objetivo: Máxima flexibilidad en el diseño de los componentes de negocio.	→ System.Transactions (A partir de .NET 2.0)
- Un único Servidor SGBD antiguo (Tipo SQL Server 2000), para las mismas transacciones - Objetivo: Máximo rendimiento en transacciones locales	→ Transacciones ADO.NET
- 'n' Servidores SGBD y Fuentes de Datos Transaccionales para Transacciones Distribuidas - Objetivo: Máxima integración con otros entornos Transaccionales (Transacciones HOST, MSMQ, etc.)	→ System.Transactions (A partir de .NET 2.0) → Enterprise Services (COM+) se podría utilizar también, pero es tecnología más antigua relacionada con COM+ y componentes COM.

- Cualquier SGBD y ejecución de una transacción concreta muy crítica en cuanto a su rendimiento máximo. - Objetivo: Máximo rendimiento absoluto, aun cuando se rompa con reglas de diseño en N-Capas	→ Transacciones internas con Transact-SQL

Así pues, como norma general y salvo excepciones, la mejor opción es **System.Transactions**.

Tabla 6.- Guía de Arquitectura Marco

Regla Nº: I8.

El sistema de gestión de transacciones a utilizar por defecto en .NET será 'System.Transactions'

<u>Norma</u>

- El sistema de implementación de transacciones más potente y flexible en .NET es **System.Transactions**. Ofrece aspectos, como 'transacciones promocionables', y máxima flexibilidad al soportar transacciones locales y transacciones distribuidas.

- Para la mayoría de las transacciones de una aplicación N-Layer, la recomendación es hacer uso del **Modelo implícito** de **System.Transactions**, es decir, utilizando '**TransactionScope'**. Aunque este modelo no llega al mismo nivel de rendimiento que las transacciones manuales o explícitas, son la forma más fácil y transparente de desarrollar, por lo que se adaptan muy bien a las Capas del Dominio. Si no se quiere hacer uso del Modelo Implícito (TransactionScope), se puede entonces hacer uso del Modelo Manual utilizando la clase '**Transaction**'del *namespace* **System.Transactions**. Considerarlo en casos puntuales o con transacciones más pesadas.

Referencias

ACID Properties (Propiedades ACID)
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconacidproperties.asp

System.Transactions
http://msdn.microsoft.com/en-us/library/system.transactions.aspx

4.2.2.- Implementación de Transacciones en la Capa de Servicios del Dominio

El inicio y coordinación de las transacciones, siguiendo un diseño correcto, normalmente se realizará en la capa de SERVICIOS de los componentes de APLICACIÓN (También es factible en la capa de Dominio, según preferencias, pero en la presente guía´, y como hemos explicado, proponemos realizar toda la coordinación de 'fontanería' como el uso de Repositorios y UoW desde la capa de aplicación, para dejar mucho más limpia a la Capa de Dominio solo con lógica de negocio).

Cualquier diseño de aplicación con transacciones de negocio, deberá incluir en su implementación, una gestión de transacciones, de forma que se pueda realizar una **secuencia de operaciones como una sola unidad de trabajo a ser aplicada o revocada completa y unitariamente si se produce algún error**.

Toda aplicación en N-Capas debería poder tener la capacidad de establecer las transacciones a nivel de los componentes de aplicación o negocio y no embeberlo dentro de la capa de datos), como se muestra en el siguiente esquema:

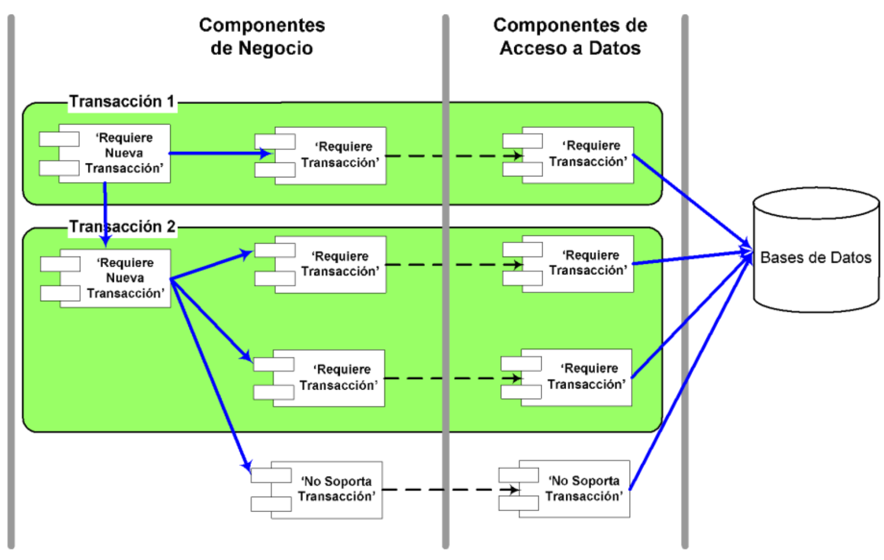


Figura 10.- Esquema Transacciones a nivel de componentes

Toda transacción nativa de tipo estándar (no transacción compensatoria) debe cumplir las propiedades **ACID**:

- **Atomicity (Atomicidad):** Una transacción debe ser una unidad atómica de trabajo, es decir, o se hace todo o no se hace nada.

- **Consistency (Consistencia):** Debe dejar los datos en un estado consistente y coherente una vez realizada la transacción.

- **Isolation (Aislamiento):** Las modificaciones realizadas por transacciones son tratadas de forma independiente, como si fueran un solo y único usuario de la base de datos

- **Durability (Durabilidad):** Una vez concluida la transacción sus efectos serán permanentes y no habrá forma de deshacerlos.

4.2.3.- Modelo de Concurrencia en actualizaciones y transacciones

Es importante identificar el modelo de concurrencia apropiado y determinar cómo se gestionarán las transacciones. Para la concurrencia, se puede elegir entre un modelo optimista o un modelo pesimista. Con el modelo de concurrencia optimista, no se mantienen bloqueos en las fuentes de datos pero las actualizaciones requieren de cierto código de comprobaciones, normalmente contra una foto o '*timestamp*' para comprobar que los datos a modificar no han cambiado en origen (B.D.) desde la última vez que se obtuvieron. Con el modelo de concurrencia pesimista, los datos se bloquean y no se pueden actualizar por ninguna otra operación hasta que dichos datos estén desbloqueados. El modelo 'pesimista' es bastante típico de aplicaciones Cliente/Servidor donde no se tiene un requerimiento de soportar una gran escalabilidad de usuarios concurrentes (p.e. miles de usuarios concurrentes). Por el contrario, **el modelo de 'concurrencia optimista' es mucho más escalable por no mantener un nivel tan alto de bloqueos en la base de datos y es por lo tanto el modelo a elegir normalmente por la mayoría de aplicaciones Web, N-Tier, y SOA**.

Tabla 7.- Guía de Arquitectura Marco

Regla Nº: I9.	**El modelo de concurrencia, por defecto, será 'Concurrencia Optimista'.**

Norma

- El modelo de concurrencia en aplicaciones N-Layer DDD con tipología de despliegue Web, N-Tier o SOA, será modelo de 'Concurrencia Optimista'.
  A nivel de implementación, es mucho más sencillo realizar una implementación de gestión de excepciones de 'Concurrencia Optimista' con las entidades 'Self-Tracking' de Entity Framework.
  Por supuesto, si se identifican razones de peso, en casos concretos, para hacer uso del modelo de concurrencia pesimista, se deberá de hacer, pero normalmente como una excepción.

Ventajas

- Mayor escalabilidad e independencia de las fuentes de datos
- Menor volumen de bloqueos en base de datos que el modelo 'pesimista'.
- Para aplicaciones de escalabilidad a "volumen Internet", es mandatorio este tipo de modelo de concurrencia.

Desventajas

- Mayor esfuerzo en desarrollo para gestionar las excepciones, si no se dispone de ayuda adicional como las entidades '*Self-Tracking'* de *Entity Framework.*
- En operaciones puntuales donde el control de concurrencia y orden de operaciones es crítico y no se desea depender de decisiones del usuario final si se producen excepciones, el modelo de concurrencia pesimista siempre ofrece un control de concurrencia más férreo y restrictivo.
- Si la posibilidad de conflicto de datos por trabajo de usuarios concurrentes es muy alta, considerar entonces la concurrencia pesimista para evitar un número muy alto de excepciones a ser decididas por los usuarios finales.

4.2.4.- Tipos de Aislamiento de Transacciones

Utilizar un nivel de aislamiento apropiado de la transacción. Hay que balancear la consistencia versus la contención. Es decir, un nivel alto de aislamiento de la transacción ofrecerá un alto nivel de consistencia de datos pero cuesta un nivel de bloqueos mayor. Por el contrario, un nivel de aislamiento de transacción más bajo, mejorará el rendimiento global al bajar la contención pero el nivel de consistencia puede ser menor.

Así pues, a la hora de realizar una transacción, es importante conocer los distintos tipos de aislamiento de los cuales disponemos de forma que podamos aplicar aquel que resulte más óptimo para la operación que deseamos realizar. Estos son los más comunes:

- **Serialized**: Los datos leídos por la transacción actual no podrán ser modificados por otras transacciones hasta que la transacción actual no finalice. Ningún nuevo dato puede ser insertado durante la ejecución de esta transacción.

- **Repeatable Read**: Los datos leídos por la transacción actual no podrán ser modificados por otras transacciones hasta que la transacción actual no finalice. Cualquier nuevo dato podrá ser insertado durante la ejecución de esta transacción.

- **Read Committed**: Una transacción no puede leer los datos que estén siendo modificados por otra transacción si esta no es de confianza. Este es el nivel de aislamiento por defecto de un Servidor Microsoft SQL Server y de Oracle.

- **Read Uncommitted**: Una transacción puede leer cualquier dato, aunque estos estén siendo modificados por otra transacción. Este es el menor nivel de aislamiento posible, si bien permite una mayor concurrencia de los datos.

Tabla 8.- Guía de Arquitectura Marco

Regla Nº: I10.	**El Nivel de aislamiento deberá considerarse en cada aplicación y área de aplicación. Los más comunes son 'Read-Commited' ó 'Serialized'.**

○ <u>**Recomendación**</u>

En los casos en los que la transacción tenga un nivel importante de criticidad, se recomienda usar el nivel '**Serialized'**, aunque hay que ser

consciente que este nivel provocará un descenso del rendimiento así como aumentará la superficie de bloqueo en base de datos.

En cualquier caso, el nivel de aislamiento de las transacciones es algo a analizar dependiendo de cada caso particular de una aplicación.

Considerar las siguientes guías cuando se diseñan e implementan transacciones:

- Tener en cuenta cuales son las fronteras de las transacciones y habilitarlas solo cuando se necesitan. Las consultas normalmente no requerirán transacciones explícitas. También conviene conocer el nivel de aislamiento de transacciones que tenga la base de datos. Por defecto SQL Server ejecuta cada sentencia individual SQL como una transacción individual (Modo transaccional *auto-commit)*.

- Las transacciones deben ser en el tiempo lo más cortas posibles para minimizar el tiempo que se mantienen los bloqueos en las tablas de la base de datos. Evitar también al máximo posible los bloqueos en datos compartidos pues pueden bloquear el acceso a otro código. Evitar el uso de bloqueos exclusivos pues pueden originar interbloqueos.

- Hay que evitar bloqueos en transacciones de larga duración. En dichos casos en los que tenemos procesos de larga duración pero nos gustaría que se comporte como una transacción, implementar métodos compensatorios para volver los datos al estado inicial en caso de que una operación falle.

A continuación, a modo ilustrativo, se muestra un ejemplo de un método en una clase de SERVICIO del Dominio (**BankTransferService**) que inicia una transacción involucrando a operaciones de Servicios del Dominio, Entidades del Dominio y Repositorios respectivos para persistir los cambios de la operación:

C#

***Namespace* de los Servicios de Capa Aplicación en un módulo ejemplo**

```
…
namespace Microsoft.Samples.NLayerApp.Application.MainModule.BankingManagement
{
```

Servicio del Dominio

Contrato/Interfaz a cumplir

```
    public class BankingManagementService:IBankingManagementService
    {
        IBankTransferDomainService _bankTransferDomainService;
        IBankAccountRepository _bankAccountRepository;
```

Constructor con Inyección de Dependencias

```
        public BankingManagementService(IBankTransferDomainService 
bankTransferDomainService, IBankAccountRepository bankAccountRepository)
```

```
        {
            _bankTransferDomainService = bankTransferDomainService;
            _bankAccountRepository = bankAccountRepository;
        }
```

Método de Servicio App para realizar Transacción

```
        public void PerformTransfer(string fromAccountNumber, string 
toAccountNumber, decimal amount)
        {
            //Process: 1° Start Transaction
            //         2° Get Accounts objects from Repositories
            //         3° Call PerformTransfer method in Domain Service
            //         4° If no exceptions, save changes using repositories 
and Commit Transaction

            //Create a transaction context for this operation
            TransactionOptions txSettings = new TransactionOptions()
            {
                Timeout = TransactionManager.DefaultTimeout,
                IsolationLevel = IsolationLevel.Serializable
            };
```

Tipo de Aislamiento de Transacción

Requiere una Transacción

```
        using (TransactionScope scope = new 
TransactionScope(TransactionScopeOption.Required, txSettings))
            {
```

Patrón UoW (Unit of work) para operaciones con Repositorios

```
                //Get Unit of Work
                IUnitOfWork unitOfWork = _bankAccountRepository.StoreContext 
as IUnitOfWork;
```

Creación Especificación de consulta

```
                //Create Queries' Specifications
                BankAccountNumberSpecification originalAccountQuerySpec = new 
BankAccountNumberSpecification(fromAccountNumber);
                BankAccountNumberSpecification destinationAccountQuerySpec = 
new BankAccountNumberSpecification(toAccountNumber);
```

Obtención de entidades y datos requeridos para la transferencia

```
                //Query Repositories to get accounts
                BankAccount originAccount = 
_bankAccountRepository.GetBySpec(originalAccountQuerySpec as 
ISpecification<BankAccount>).SingleOrDefault();

                BankAccount destinationAccount = 
_bankAccountRepository.GetBySpec(destinationAccountQuerySpec as 
ISpecification<BankAccount>).SingleOrDefault();

                ////Start tracking STE entities (Self Tracking Entities)
                originAccount.StartTrackingAll();
                destinationAccount.StartTrackingAll();
```

Llamar a Operaciones del Dominio para Transferencia

```
                //Excute Domain Logic for the Transfer (In Domain Service)
                _bankTransferDomainService.PerformTransfer(originAccount, 
destinationAccount, amount);
```

```
                    //Save changes and commit operations.
                    //This opeation is problematic with concurrency.
                    //"balance" propety in bankAccount is configured
                    //to FIXED in "WHERE concurrency checked predicates"
```

Uso de Repositorios: 'Marcado' para Actualización

```
                    _bankAccountRepository.Modify(originAccount);
                    _bankAccountRepository.Modify(destinationAccount);
```

Commit de Unit of Work. La B.D. se actualiza en este momento

```
                    //Complete changes in this Unit of Work
                    unitOfWork.CommitAndRefreshChanges();
```

Commit de Transacción

```
                    //Commit the transaction
                    scope.Complete();
                }
            }
        }
    }
```

Algunas consideraciones sobre el ejemplo anterior:

- Como se puede observar, en este Servicio de Capa de Aplicación es donde implementamos toda la <u>coordinación de 'fontanería'</u>, es decir, creación de transacción y configuración de su tipo, uso de 'Unit of Work', llamadas a Repositorios para obtener entidades y para persistirlas finalmente, etc. y en definitiva toda la coordinación necesaria de la aplicación pero aspectos que no discutiríamos con un experto de negocio/dominio. En cambio, toda la lógica del Dominio (las operaciones de la transferencia bancaria) son las que quedan encapsuladas en el Servicio del Dominio y la lógica de negocio de las propias entidades (En el ejemplo, la entidad BankAccount y el servicio de Dominio BankTransferDomainService).

- Por el hecho de emplear **using**, no es necesario gestionar manualmente el *rollback* de la transacción. Cualquier excepción que se produzca al insertar alguna de las regiones, provoca que se aborte la transacción.

- Los **UoW** (*Unit of work*) facilitan un contexto donde los Repositorios apuntan/registran las operaciones de persistencia que quieren hacerse, pero realmente no se efectúan (todos los cambios simultáneamente) hasta que de forma explícita llamamos a 'unitOfWork.CommitAndRefreshChanges()'.

Anidamiento de transacciones

System.Transactions permite el anidamiento de transacciones de forma transparente. Un ejemplo típico es tener otro "TransactionScope" dentro de un método interno (Por ejemplo en uno de los métodos de la clase "*BankAccount*", etc.). La transacción

original se extenderá con el nuevo TransactionScope de una forma u otra dependiendo del 'TransactionScopeOption' especificado en el TransactionScope interno.

Como se ve, la ventaja de este modelo es su flexibilidad y facilidad para el desarrollo.

Tabla 9.- Guía de Arquitectura Marco

Regla Nº: I11.

El tipo de TransactionScope por defecto será 'Required'.

○ **Recomendación**

- Si no se especifica un *scope* de transacción en los Servicios 'hoja', es decir, los que ya hacen uso de REPOSITORIOS, entonces sus operaciones se enlistarán a transacciones de más alto nivel que pudieran hacer sido creadas. Pero si en estos SERVICIOS 'hoja', implementamos también un *TransactionScope,* normalmente es recomendable que se configure como '*Required*'.

  Esto es así para que en caso de llamarse al Servicio con nuestra transacción desde código que todavía no ha creado ninguna transacción, entonces se creara una transacción nueva con las operaciones correspondientes. Pero si se le llama desde otra clase/servicio que ya tiene creada una transacción, esta llamada simplemente debería ampliar a la transacción actual, entonces, estando como '*Required'* (TransactionScopeOption.Required) se enlistará correctamente a dicha transacción existente. Por el contrario, si estuviera como '*RequiredNew'*, aunque ya exista una transacción inicial, al llamar a este componente, se creará otra transacción nueva. Por supuesto, todo esto depende de las reglas de negocio concretas. En algunos casos puede interesar este otro comportamiento.

  Esta configuración de la transacción se implementa mediante la sintaxis de ***'TransactionScope()***' de System.Transactions.

Referencias

Introducing System.Transactions in the .NET Framework 2.0:
http://msdn2.microsoft.com/en-us/library/ms973865.aspx

Concurrency Control at **http://msdn.microsoft.com/enus/library/ms978457.aspx**.

Integration Patterns at **http://msdn.microsoft.com/enus/library/ms978729.aspx**.

4.3.- Implementación de pruebas en la capa de Aplicación

Las pruebas de la capa de aplicación normalmente deberán realizar *testing* especialmente sobre los Servicios de aplicación.

Las pruebas de los servicios de aplicación son relativamente complejas puesto que involucran dependencias de otros elementos como por ejemplo el *IContext* utilizado o bien otros servicios (de aplicación o dominio) y por supuesto invocación a lógica de entidades del dominio.

C#

```
   [TestClass()]

[DeploymentItem("Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule.
Mock.dll")]

[DeploymentItem("Microsoft.Samples.NLayerApp.Infrastructure.Data.MainModule.
dll")]
       public class BankingManagementServiceTests
       {

           [TestMethod()]
           public void PerformTransfer_Invoke_Test()
           {
               //Arrange

               IBankingManagementService bankTransfersService =
ServiceFactory.Current.Resolve<IBankingManagementService>();
               IBankingManagementService bankAccountService =
ServiceFactory.Current.Resolve<IBankingManagementService>();

               string bankAccountFrom = "BAC0000001";
               string bankAcccountTo = "BAC0000002";
               decimal amount = 10M;
               decimal actualBanlance = 0M;

               //Act

               //find actual balance in to account
               actualBanlance =
bankAccountService.FindBankAccountByNumber(bankAcccountTo).Balance;

               bankTransfersService.PerformTransfer(bankAccountFrom,
bankAcccountTo, amount);

               //Assert
               //check balance
               decimal balance =
bankAccountService.FindBankAccounts(bankAcccountTo,
null).SingleOrDefault().Balance;
               Assert.AreEqual(actualBanlance + amount, balance);
           }
```

Por supuesto, el archivo de configuración del contendor de dependencias puede incluir la posibilidad, al igual que hacemos en la capa de infraestructura de la persistencia, de incorporar una simulación de la interfaz *IContext*, es decir, hacer que las pruebas se ejecuten finalmente contra una base de datos real o no, lo cual influye en gran medida en la velocidad de los tests, recuerde aquí un conocido anti-patrón en pruebas unitarias, **SlowTest,** que es de vital importancia si queremos que los desarrolladores no dejen de pasar pruebas debido a la lentitud de las mismas.

En el caso concreto de nuestra aplicación ejemplo NLayerApp (en CODEPLEX), este cambio para que se ejecuten las pruebas contra estructuras en memoria en lugar de contra la base de datos, es configurable desde el Web.config del proyecto de servicios WCF:

Web.config de proyecto hosting de WCF en aplicación ejemplo NLayerApp

```
<appSettings>
  <!--RealAppContext - Real Container-->
  <!--FakeAppContext - Fake Container-->
  <!--<add key="defaultIoCContainer" value="FakeAppContext" />-->
  <add key="defaultIoCContainer" value="RealAppContext" />
</appSettings>
```

Internamente, se está haciendo *mocking* del contexto de Entity Framework contra un entorno simulado de estructuras en memoria. Al no acceder a la base de datos, las pruebas unitarias se ejecutarán mucho más rápidamente, especialmente notable cuando tengamos muchos cientos o incluso miles de pruebas unitarias.

CAPÍTULO

7

Capa de Servicios Distribuidos

1.- SITUACIÓN EN ARQUITECTURA N-CAPAS

Esta sección describe el área de arquitectura relacionada con esta capa, que lógicamente es una Arquitectura Orientada a Servicios, que se solapa en gran medida con SOA (*Service Oriented Architecture*).

NOTA IMPORTANTE:
En el presente capítulo, cuando hacemos uso del término 'Servicio', nos estaremos refiriendo, por defecto, a 'Servicios Distribuidos', a Servicios-Web, no a Servicios internos de Capas del Dominio/Aplicación/Infraestructura según conceptos DDD.

En el siguiente diagrama se muestra cómo encaja típicamente esta capa (Servicios Distribuidos), dentro de nuestra '*Arquitectura N-Capas Orientada al Dominio*':

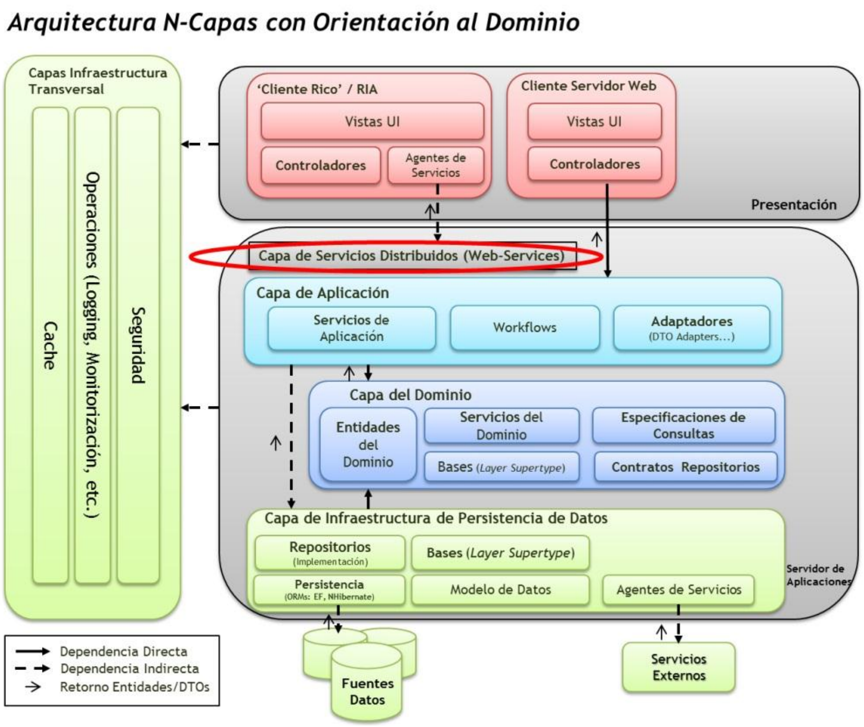


Figura 1.- Situación de Capa de Servicios Distribuidos

La Capa de Servicios normalmente incluye lo siguiente:

- **Interfaces/Contratos de Servicios**: Los Servicios exponen un interfaz de servicio al que se envían los mensajes de entrada. En definitiva, los servicios son como una fachada que expone la lógica de aplicación y del dominio a los consumidores potenciales, bien sea la Capa de Presentación o bien sean otros Servicios/Aplicaciones remotas.

- **Mensaje de Tipos**: Para intercambiar datos a través de la capa de Servicios, es necesario hacerlo mediante mensajes que envuelven a estructuras de datos. La capa de servicio también incluirá tipos de datos y contratos que definan los tipos de datos utilizados en los mensajes.

SOA, sin embargo, abarca mucho más que el diseño e implementación de una Capa de Servicios distribuidos interna para una única aplicación N-Layer. La virtud de SOA es precisamente el poder compartir ciertos Servicios/Aplicaciones y dar acceso a ellos de una forma estándar, pudiendo realizar integraciones de una forma interoperable que hace años eran costosísimas.

Antes de centrarnos en el diseño de una Capa de Servicios dentro de una aplicación N-Layer, vamos a realizar una introducción a SOA.

2.- ARQUITECTURAS ORIENTADAS A SERVICIOS Y ARQUITECTURAS EN N-CAPAS (N-LAYER)

Es importante destacar que las tendencias de arquitecturas orientadas a servicios (**SOA**) no son antagónicas a arquitecturas ***N-Layered (N-Capas)***, por el contrario, son arquitecturas que se complementan unas con otras. **SOA** es una arquitectura de alto nivel que define 'como' intercomunicar unas aplicaciones (Servicios) con otras. Y simultáneamente, cada una de dichas **aplicaciones/servicios SOA** pueden estar internamente estructuradas siguiendo patrones de diseño de ***Arquitecturas N-Layer***.

SOA trata de definir 'buses de comunicación estándar' y corporativos entre las diferentes aplicaciones/servicios de una empresa, e incluso entre servicios de diferentes empresas en diferentes puntos de Internet.

En el siguiente gráfico se muestra un ejemplo básico de "*Bus de comunicación estándar SOA*" entre diferentes aplicaciones/Servicios de una empresa:

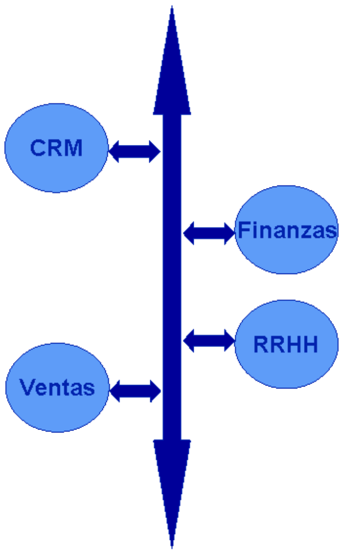


Figura 2.- Bus de comunicación estándar SOA

Cada Servicio/Aplicación SOA tendrá necesariamente una implementación interna donde esté articulada la lógica de negocio, accesos a datos y los propios datos (estados) de la aplicación/servicio. Y toda la comunicación que entra/salga del servicio serán siempre mensajes (mensajes SOAP, etc.).



Figura 3.- Vista interna de un servicio distribuido

Esta implementación interna es la que normalmente (haciéndolo de una forma estructurada) se realiza siguiendo patrones de diseño de Arquitecturas lógicas en N-Capas (**N-Layer**) y distribución física (*deployment* en servidores) según arquitecturas en N-Niveles **(N-Tier)**.

De igual forma, la estructura en capas que podría tener dicho Servicio SOA, podría ser una estructura en capas alineada con la Arquitectura en capas que proponemos en la presente guía, es decir, una Arquitectura N-Capas Orientada al Dominio, con tendencias de Arquitectura según DDD. Este punto lo explicamos en más detalle más adelante.

3.- SITUACIÓN DE ARQUITECTURA N-LAYER CON RESPECTO A APLICACIONES AISLADAS Y A SERVICIOS SOA

La arquitectura interna de un Servicio SOA puede ser por lo tanto muy similar a la de una aplicación aislada, es decir, implementando la arquitectura interna de ambos (Servicio SOA y Aplicación aislada) como una ***Arquitectura N-Layer*** (diseño de Arquitectura lógica en N-Capas de componentes).

La principal diferencia entre ambos es que un Servicio SOA es, visto 'desde fuera' (desde otra aplicación), como algo 'no visual'. Por el contrario, una aplicación aislada tendrá además una **Capa de Presentación** (es decir, la parte 'cliente' de la aplicación a ser utilizada visualmente por el usuario final).

Es importante resaltar que **una aplicación 'independiente y visual' también puede ser simultáneamente un Servicio SOA para publicar (dar acceso) a sus componentes y lógica de negocio a otras aplicaciones externas**.

El orden que vamos a seguir en este documento es explicar primero las bases de la Arquitectura SOA. Posteriormente se explicará la implementación de Servicios Distribuidos con **WCF** (*Windows Communication Foundation)*.

4.- ¿QUÉ ES SOA?

SOA (*Service Oriented Architecture*) ó 'Service Orientation' es complementario a la orientación a objetos (OOP) y aplica aspectos aprendidos a lo largo del tiempo en el desarrollo de software distribuido.

Las razones de aparición de SOA son básicamente las siguientes:

- La Integración entre aplicaciones y plataformas es difícil
- Existen sistemas heterogéneos (diferentes tecnologías)
- Existen múltiples soluciones de integración, independientes y ajenas unas a otras.

Se necesita un planteamiento estándar que aporte:

- Arquitectura orientada a servicios
- Basada en un "bus común de mensajería"
- Estándares para todas las plataformas

SOA trata de definir 'buses de comunicación estándar' y corporativos entre las diferentes aplicaciones/servicios de una empresa, e incluso entre servicios de diferentes empresas en diferentes puntos de Internet.

La '*Orientación a Servicios*' se diferencia de la '*Orientación a Objetos*' primeramente en cómo define el término '*aplicación*'. El 'Desarrollo Orientado a Objetos' se centra en aplicaciones que están construidas basadas en librerías de clases interdependientes. SOA, sin embargo, hace hincapié en sistemas que se construyen basándose en un conjunto de servicios autónomos. Esta diferencia tiene un profundo impacto en las asunciones que uno puede hacer sobre el desarrollo.

Un 'servicio' es simplemente un programa con el que uno interactúa mediante mensajes. Un conjunto de servicios instalados/desplegados sería un 'sistema'. Los servicios individuales se deben de construir de una forma consistente (disponibilidad y estabilidad son cruciales en un servicio). Un sistema agregado/compuesto por varios servicios se debe construir de forma que permita el cambio y evolución de dichos servicios, el sistema debe adaptarse a la presencia de nuevos servicios que aparezcan a lo largo del tiempo después de que se hubieran desplegado/instalado los servicios y

clientes originales. Y dichos cambios no deben romper la funcionalidad del sistema actual.

Otro aspecto a destacar es que un Servicio-SOA debe de ser, por regla general, interoperable. Para eso, debe de basarse en especificaciones estándar a nivel de **protocolos**, **formato** de datos serializados en las comunicaciones, etc.

Actualmente existen dos tendencias de Arquitectura en cuanto a Servicios Web:

- SOAP (Especificaciones WS-I, WS-*)
- REST (Servicios RESTful)

SOAP está basado especialmente en los mensajes SOAP, en un formato de los mensajes que es **XML** y **HTTP** como protocolo de transporte (como los servicios-web **ASMX** o servicios **WCF** con *binding* ***BasicHttpBinding (Basic Profile)*** ó ***WsHttpBinding (WS-*)***).

REST está muy orientado a la URI, al direccionamiento de los recursos basándonos en la URL de HTTP y por lo tanto los mensajes a intercambiar son mucho más sencillos y ligeros que los mensajes XML de SOAP.

A nivel de tecnología, como extenderemos en el capítulo de implementación, con ***WCF*** **(*Windows Communication Foundation*)** se nos permite también otros tipos de formatos de datos y protocolos de transporte que no son interoperables, solamente compatibles con extremos .NET (como ***NetTcpBinding***. ***NetNamedPipeBinding***, ó ***NetPeerTcpBinding***). Pueden ser muy útiles como protocolos de comunicaciones remotas **dentro de una misma aplicación/servicio**, pero no son los más adecuados para Servicios-SOA interoperables.

5.- PILARES DE SOA ('SERVICE ORIENTATION TENETS')

Siguiendo la visión de SOA tradicional de Microsoft, el desarrollo orientado a servicios está basado en los siguientes cuatro pilares, los cuales fueron introducidos hace algunos años, especialmente por **Don Box**, uno de los precursores de SOAP:

Tabla 1.- Los cuatro pilares SOA

'Service Orientation Tenets'
1.- Las fronteras de los Servicios deben ser explícitas
2.- Los Servicios deben ser Autónomos
3.- Los Servicios deben compartir Esquemas y Contratos, no Clases y Tipos
4.- La Compatibilidad se debe basar en Políticas

A continuación pasamos a explicar cada uno de estos puntos base de SOA.

Las fronteras de los Servicios deben ser explícitas: Una aplicación orientada a servicios a menudo está compuesta por varios servicios distribuidos en diferentes puntos geográficos distantes, múltiples autoridades de confianza, y diferentes entornos de ejecución. El coste de traspasar dichas fronteras no es trivial en términos de complejidad y especialmente de rendimiento (la latencia existente en cualquier comunicación remota siempre tiene un coste; si el formato de los mensajes es XML-SOAP y el protocolo es HTTP, este 'coste' en rendimiento es aún mayor).

Los diseños SOA reconocen estos costes recalcando que hay un coste en el momento de cruzar dichas fronteras, por lo que lógicamente, este hecho debe minimizarse en la medida de lo posible.

Debido a que cada comunicación que cruce dichas fronteras tiene un coste potencial, la orientación a servicios se basa en un modelo de intercambio de mensajes explícito en lugar de un sistema de invocación remota de métodos de forma implícita.

Aunque SOA soporta la notación 'estilo-RPC' (invocación síncrona de métodos), también puede soportar comunicación asíncrona de mensajes y al mismo tiempo asegurar el orden de llegada de dichos mensajes asíncronos, y poder indicar de forma explícita a qué cadena de mensajes pertenece un mensaje en particular. Esta indicación explícita es útil para correlaciones de mensajes y para implementar modelos de concurrencia.

El concepto de que las 'fronteras son explícitas' se aplica no solamente a la comunicación entre diferentes servicios, también incluso a la comunicación entre desarrolladores como personas. Incluso en escenarios en los que los servicios se despliegan en un único punto, puede ser común que los desarrolladores del mismo sistema estén situados en diferentes situaciones geográficas, culturales y/o con fronteras organizacionales. Cada una de dichas fronteras incrementa el coste de comunicación entre los desarrolladores. La 'Orientación a Servicios' se adapta a este modelo de 'desarrollo distribuido' reduciendo el número y complejidad de abstracciones que deban ser compartidas por los desarrolladores a lo largo de las fronteras de servicios. Si se mantiene el 'área de superficie' de un servicio tan pequeña como sea posible, la interacción y comunicación entre las organizaciones de desarrollo se reducen.

Un aspecto que es importante en los diseños orientados a servicios es que la **simplicidad** y generalización no son un 'lujo' sino más bien una aspecto crítico de 'supervivencia.

Por último y relacionado con la importancia de tener muy en cuenta a 'las fronteras', la idea de que puedes tomar un interfaz de un objeto local y extenderlo a lo largo de fronteras de diferentes máquinas remotas creando una transparencia en la localización (como funcionaba el antiguo DCOM), es falsa y en muchos casos dañina. Aunque es cierto que tanto los objetos remotos como los objetos locales tienen el mismo interfaz desde la perspectiva del proceso que lo consume, el comportamiento del interfaz llamado es muy diferente dependiendo de la localización. Desde la perspectiva del cliente, una implementación remota del interfaz está sujeta a latencia de

red, fallos de red, y fallos de sistemas distribuidos que no existen en implementaciones locales. Por todo esto, se debe de implementar una cantidad significativa de código de detección de errores y corrección de lógica (p.e. en los ***Agentes de Servicio***) para anticiparse a los impactos derivados del uso de interfaces de objetos remotos.

Los Servicios deben ser Autónomos: La orientación a servicios se parece al mundo real en el sentido en que no asume la presencia de un omnisciente y omnipotente oráculo que conoce y controla todas las partes de un sistema en ejecución. Esta noción de autonomía del servicio aparece en varias facetas del desarrollo, pero la más importante es ***autonomía en el área de desarrollo independiente, versionado y despliegue (tanto de código como de bases de datos)***.

Los programas orientados a objetos, normalmente se despliegan/instalan como una única unidad. A pesar de los grandes esfuerzos hechos en los años 90 para habilitar que se pudieran instalar clases de forma independiente, la disciplina requerida para habilitar interacción orientada a objetos con un componente demostró ser poco práctica para la mayoría de desarrollos de organizaciones.

Unido a las complejidades de versionados de interfaces en la orientación a objetos, muchas organizaciones se han vuelto muy conservadoras en como despliegan el código orientado a objetos. La popularidad del 'despliegue XCOPY' de .NET framework es un indicador de este punto.

El desarrollo orientado a servicios comienza a partir de la orientación a objetos, asumiendo que la instalación atómica de una aplicación es realmente la excepción, no la regla. Mientras los servicios individuales se instalan normalmente de forma atómica, el estado de despliegues/instalaciones agregadas de la mayoría de sistemas y aplicaciones raramente lo son. Es común que un servicio individual sea instalado mucho tiempo antes de que una aplicación que lo consuma sea ni siquiera desarrollada y posteriormente desplegada.

También es común en la topología de aplicaciones orientadas a servicios que los sistemas y servicios evolucionen a lo largo del tiempo, algunas veces sin intervención directa de un administrador o desarrollador. El grado en el cual se pueden introducir nuevos servicios en un sistema orientado a servicios depende tanto de la complejidad de las interacciones de los servicios como de la *ubicuidad* (poder ser encontrado y explorado) de los servicios que interaccionen de la misma forma (que tengan una misma funcionalidad inicial).

La orientación a servicios recomienda un modelo que incremente la ***ubicuidad*** (poder ser encontrado y explorado, por ejemplo mediante ***UDDI*** y ***WSDL***), reduciendo la complejidad de las interacciones de los servicios.

La noción de servicios autónomos también impacta en la forma en que las excepciones y errores se gestionan. Los objetos se despliegan para ejecutarse en el mismo contexto de ejecución que la aplicación que los consume. Sin embargo, los diseños orientados a servicios asumen que esa situación es una excepción, no la regla. Por esa razón, los servicios esperan que la aplicación que los consume (aplicación cliente) pueda fallar sin notarlo y a menudo sin notificarlo. Para mantener integridad de sistema, los diseños orientados a servicios hacen uso de técnicas para tratar con modos

parciales de fallos. Técnicas como transacciones, colas persistentes y despliegues redundantes y *clusters* son bastante comunes en sistemas orientados a servicios.

Debido a que muchos servicios se despliegan para que funcionen en redes públicas (como Internet), SOA asume que no solamente los mensajes que lleguen pueden estar mal-formados sino que también pueden haber sido modificados y transmitidos con propósitos maliciosos (La seguridad es muy importante en los servicios).

SOA se protege a si mismo estableciendo pruebas en todos los envíos de mensajes requiriendo a las aplicaciones que prueben que todos los derechos y privilegios necesarios los tienen concedidos.

De forma consistente con la noción de autonomía de servicios, SOA se basa completamente en relaciones de confianza (por ejemplo ***WS-Federation***) gestionadas administrativamente para poder evitar mecanismos de autenticación por servicio, algo común por el contrario en aplicaciones Web clásicas.

Los Servicios deben compartir Esquemas y Contratos, no Clases y Tipos:

La programación orientada a objetos recomienda a los desarrolladores el crear nuevas abstracciones en forma de clases. La mayoría de los entornos de desarrollo modernos no solamente hacen sencillo el definir nuevas clases, sino que los IDEs modernos incluso guían al desarrollador en el proceso de desarrollo según el número de clases aumenta (características como *IntelliSense*, etc.). Las clases son abstracciones convenientes porque comparten estructura y comportamiento en una única unidad específica. SOA sin embargo no recomienda construir exactamente así. En lugar de esa forma, los servicios interaccionan basándose solamente en ***esquemas*** (para estructuras de datos) y ***contratos*** (para comportamientos). Cada servicio muestra un contrato que describe la estructura de mensajes que puede mandar y/o recibir así como algunos grados de restricciones de aseguramiento de orden en mensajes, etc. Esta separación estricta entre estructuras de datos y comportamientos simplifica mucho el desarrollo. Conceptos de objetos distribuidos como '*marshal-by-value*' requieren de una ejecución y entorno de seguridad común que está en conflicto directo con las metas de desarrollo autónomo.

Debido a que el contrato y esquema, de un servicio dado, son visibles a lo largo de largos períodos de tiempo y espacio, SOA requiere que los contratos y esquemas se mantengan estables a lo largo del tiempo. Por regla general, es imposible propagar cambios en un esquema y/o contrato a todas las partes que han consumido alguna vez un servicio. Por esa razón, el contrato y esquema utilizados en diseños SOA tienden a tener más flexibilidad que los interfaces tradicionales orientados a objetos, extendiéndose en lugar de cambiándose interfaces existente, etc.

La Compatibilidad de los servicios se debe basar en Políticas: Los diseños orientados a objetos a menudo confunden *compatibilidades estructurales* con *compatibilidades semánticas*. SOA trata con estos ejes de forma separada. La **compatibilidad estructural** está basada en el **contrato** y **esquema**, todo lo cual puede ser validado e incluso requerido. La **compatibilidad semántica** (por ejemplo requerimientos de seguridad, firma, cifrado, etc.) está basada en **sentencias explícitas de capacidades y requerimientos en forma de políticas**.

Cada servicio advierte de sus capacidades y requerimientos en forma de una expresión de política legible para el sistema. Las expresiones de políticas indican qué condiciones y garantías (llamadas en programación *assertions*) deben de soportarse para habilitar el funcionamiento normal del servicio. Por ejemplo, en WSE y WCF dichas políticas se definen normalmente de forma declarativa en los ficheros XML de configuración (.config).

6.- ARQUITECTURA INTERNA DE LOS SERVICIOS SOA

SOA pretende resolver problemas del desarrollo de aplicaciones distribuidas. Un 'Servicio' puede describirse como una aplicación que expone un interfaz basado en mensajes, encapsula datos y puede también gestionar transacciones ACID (Atómicas, consistentes, Asiladas y Perdurables), con sus respectivas fuentes de datos. Normalmente, SOA se define como un conjunto de proveedores de servicios que exponen su funcionalidad mediante interfaces públicos (que pueden estar también protegidos/securizados). Los interfaces expuestos por los proveedores de servicios pueden ser consumidos individualmente o bien agregando varios servicios y formando proveedores de servicios compuestos.

Los servicios SOA también pueden proporcionar interfaces 'estilo-RPC, si se requieren. Sin embargo, los escenarios 'petición-respuesta síncronos' deben de intentar ser evitados siempre que sea posible, favoreciendo por el contrario el consumo asíncrono de Servicios.

Los servicios se construyen internamente normalmente mediante las siguientes capas:

- **Interfaz del Servicio (Contrato)**
- **Capas de Aplicación y Dominio**
- **Acceso a datos (Infraestructura y Acceso a Datos**

En el siguiente esquema se muestra como estaría estructurado internamente el servicio ejemplo anterior:

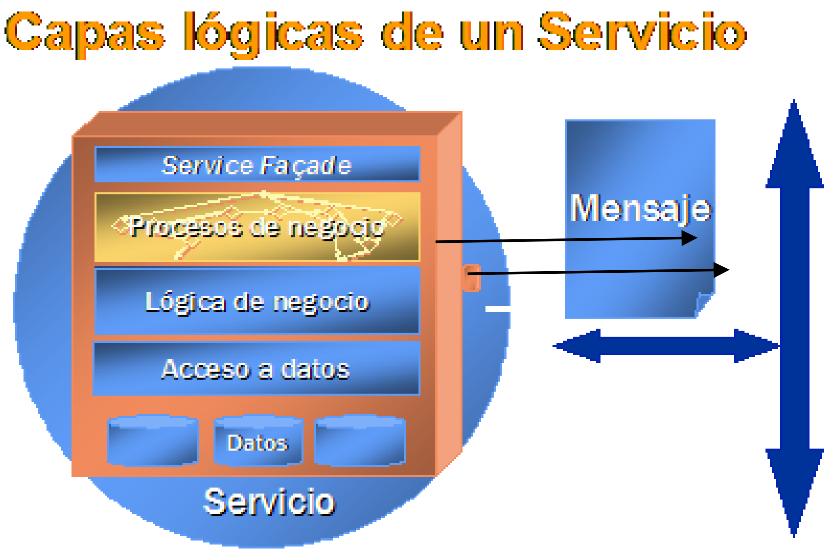


Figura 4.- Capas lógicas de un servicio

Comparado con la arquitectura interna de una aplicación N-Layer (N-Capas), es muy similar, con la diferencia de que un servicio lógicamente no tiene capa de presentación.

El 'Interfaz' se sitúa lógicamente entre los clientes del servicio y la fachada de procesos del servicio. Un único servicio puede tener varios interfaces, como un *Web-Service* basado en HTTP, un sistema de colas de mensaje (como MSMQ), un servicio-WCF con *binding* basado en TCP (puerto TCP elegido por nosotros), etc.

Normalmente un servicio distribuido debe proporcionar un interfaz 'grueso' o poco granularizado. Es decir, se intenta realizar el máximo número de acciones dentro de un método para conseguir minimizar el número de llamadas remotas desde el cliente.

En muchos casos también los servicios son '*stateless*' (sin estados y una vida de objetos internos relativa a cada llamada externa), aunque no tienen por qué ser *stateless* siempre. Un *WebService* básico (especificación WS-I) si es *stateless*, pero un servicio-WCF avanzado (especificaciones WS-* o propietarias Net), puede tener también estados y objetos compartidos, como siendo de tipo *Singleton*, *Session*, etc.).

7.- PASOS DE DISEÑO DE LA CAPA DE SERVICIOS

El mejor enfoque a la hora de diseñar un servicio consiste en comenzar por definir su contrato, el interfaz del servicio, es decir, ¿qué va a ofrecer y exponer un servicio?. Esta forma de diseñar es a lo que se conoce como "Primero el Contrato" (*Contract First*). Una vez que tenemos definido el contrato/interfaz, el siguiente paso es diseñar

la implementación, que lo que realizará es convertir los contratos de datos del servicio en entidades del dominio e interactuar también con los objetos del Dominio y Aplicación. Los pasos básicamente son:

1.- Definir los contratos de datos y mensajes que representan el esquema a utilizar por los mensajes

2.- Definir el contrato del servicio que representa las operaciones soportadas por nuestro servicio

3.- Diseñar las transformaciones de objetos, si necesitaremos realizarlas manualmente, como en el caso de transformación de DTOs a Entidades del Dominio. (Este punto puede realizarse en la Capa de Aplicación, en lugar de en la propia capa del Servicio Distribuido)

4.- Definir contratos de fallos (*Fault Contracts*) que devuelvan información de errores a los consumidores del servicio distribuido.

5.- Diseñar el sistema de integración con las Capas internas (Dominio, Aplicación, etc.). Una buena aproximación es comenzar DI (Inyección de Dependencias) es este nivel de Servicios-Web haciendo uso de la resolución de los Contenedores de IoC mayoritariamente en esta capa (Servicios Distribuidos) puesto que es el punto de entrada a la aplicación, y dejar que el sistema de IoC vaya creando todas las dependencias internas del resto de capas.

8.- TIPOS DE OBJETOS DE DATOS A COMUNICAR

Debemos determinar cómo vamos a transferir los datos de entidades a través de las fronteras físicas de nuestra Arquitectura (Tiers). En la mayoría de los casos, en el momento que queremos transferir datos de un proceso a otro e incluso de un servidor a otro, debemos serializar los datos.

Podríamos llegar a utilizar esta serialización cuando se pasa de una capa lógica a otra, pero en general esto no es una buena idea, pues tendremos penalizaciones en rendimiento.

Intentando unificar opciones, a un nivel lógico, los tipos de objetos de datos a comunicar, más comunes, a pasar de un nivel a otro nivel remoto dentro de una Arquitectura N-Tier son:

- **Valores escalares**

- **DTOs (Data Transfer Objects)**

- **Entidades del Dominio serializadas**

- **Conjuntos de registros (Artefactos desconectados)**

Todos estos tipos de objetos tienen que por supuesto poder serializarse y transmitirse por la red mediante un formato de datos tipo XML, texto con otro formato o incluso en binario.

Valores Escalares

Cuando se van a transmitir ciertamente muy pocos datos (normalmente en llamadas al servidor con parámetros de entrada), si dichos parámetros son muy pocos, es bastante normal hacer uso simplemente de valores escalares (datos sueltos de tipo *int*, *string*, etc.).

Entidades del Dominio serializadas

Cuando estamos tratando con volúmenes de datos relacionados con entidades del dominio, una primera opción (la más inmediata) es serializar y transmitir las propias entidades del dominio a la capa de presentación. Esto, dependiendo de la implementación puede ser bueno o malo. Es decir, si la implementación de las entidades está fuertemente ligada a una tecnología concreta, entonces es contrario a las recomendaciones de Arquitectura DDD, porque estamos 'contaminando' toda la arquitectura con una tecnología concreta. Sin embargo, cabe la opción de enviar entidades del dominio que sean POCO (Plain Old Clr Objects), es decir, clases serializadas cuyo código está 'bajo nuestra propiedad 100%', completamente nuestro y no dependiente de una tecnología de acceso a datos. En ese caso, la aproximación puede ser buena y muy productiva, porque podemos tener herramientas que nos generen código por nosotros para dichas clases entidad y el trabajo sea muy ágil pues incluso dichas entidades pueden realizar tareas de control de concurrencia por nosotros. Este concepto (Serializar y transmitir Entidades del Dominio a otros Tiers/Niveles físicos) lo analizaremos en el capítulo de implementación de Servicios Web.

Así pues, este enfoque (Serialización de las propias entidades del Dominio), tiene el inconveniente de dejar directamente ligado al consumidor del servicio con las entidades del dominio, las cuales podrían tener una vida de cambios a un ritmo diferente con respecto a los clientes que consumen los servicios-web. Por lo tanto, este enfoque es adecuado solo cuando se mantiene un control directo sobre la aplicación/cliente que consume los servicios-web (Como una típica Aplicación N-Tier). En caso contrario (Servicios SOA para consumidores desconocidos), es mucho más recomendable la aproximación con DTOs que explicamos a continuación.

DTOs (*Data Transfer Objects*)

Para desacoplar los clientes/consumidores de los servicios-web de la implementación interna de las Entidades del Dominio, la opción más utilizada es implementando DTOs (*Data Transfer Objects*) el cual es un patrón de diseño que consiste en empaquetar múltiples estructuras de datos en una única estructura de datos

a ser transferida entre 'fronteras físicas' (comunicación remota entre servidores y/o máquinas). Los DTOs son especialmente útiles cuando la aplicación que consume nuestros servicios tiene una representación de datos e incluso modelo que no tiene por qué coincidir completamente con el modelo de entidades del Dominio. Este patrón, por lo tanto, nos permite cambiar la implementación interna de las entidades del Dominio y siempre que se respete los interfaces de los Servicios Web y la estructura de los DTOs, dichos cambios en el servidor no afectarán a los consumidores. También permite una gestión de versiones más cómoda hacia los consumidores externos. Esta aproximación de diseño es, por lo tanto, la más apropiada cuando se tienen clientes/consumidores externos consumiendo datos de nuestros servicios-web y quien desarrolla los componentes de servidor no tiene control sobre el desarrollo de dichas aplicaciones cliente.

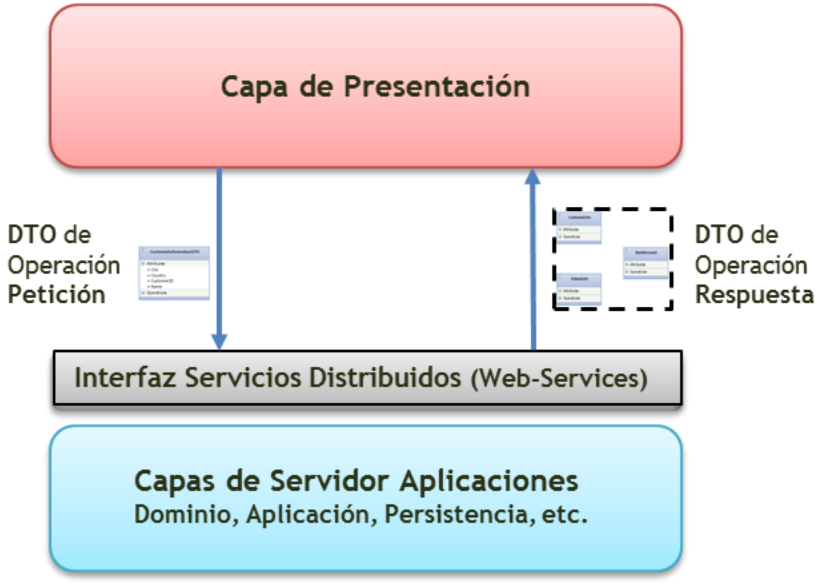


Figura 5.- Diagrama DTOs (Data Transfer Objects)

El diseño de los DTOs normalmente se intenta ajustar a las necesidades hipotéticas del consumidor (bien capa de presentación, bien otro tipo de aplicación externa) y también es importante diseñarlos para que tiendan a minimizar el número de llamadas al servicio web, mejorando así el rendimiento de la aplicación distribuida.

Para trabajar con DTOs es necesario cierta lógica de adaptación/conversión desde DTOs hacia entidades del Dominio y viceversa. Estos Adaptadores/Conversores en una Arquitectura N-Layer DDD, normalmente los situaríamos en la Capa de Aplicación, pues es un requerimiento puramente de Arquitectura de Aplicación, no del Dominio. Tampoco sería la mejor opción situarlos dentro de los propios Servicios-Web que deberían ser lo más delgados o transparentes posible en cuanto a lógica.

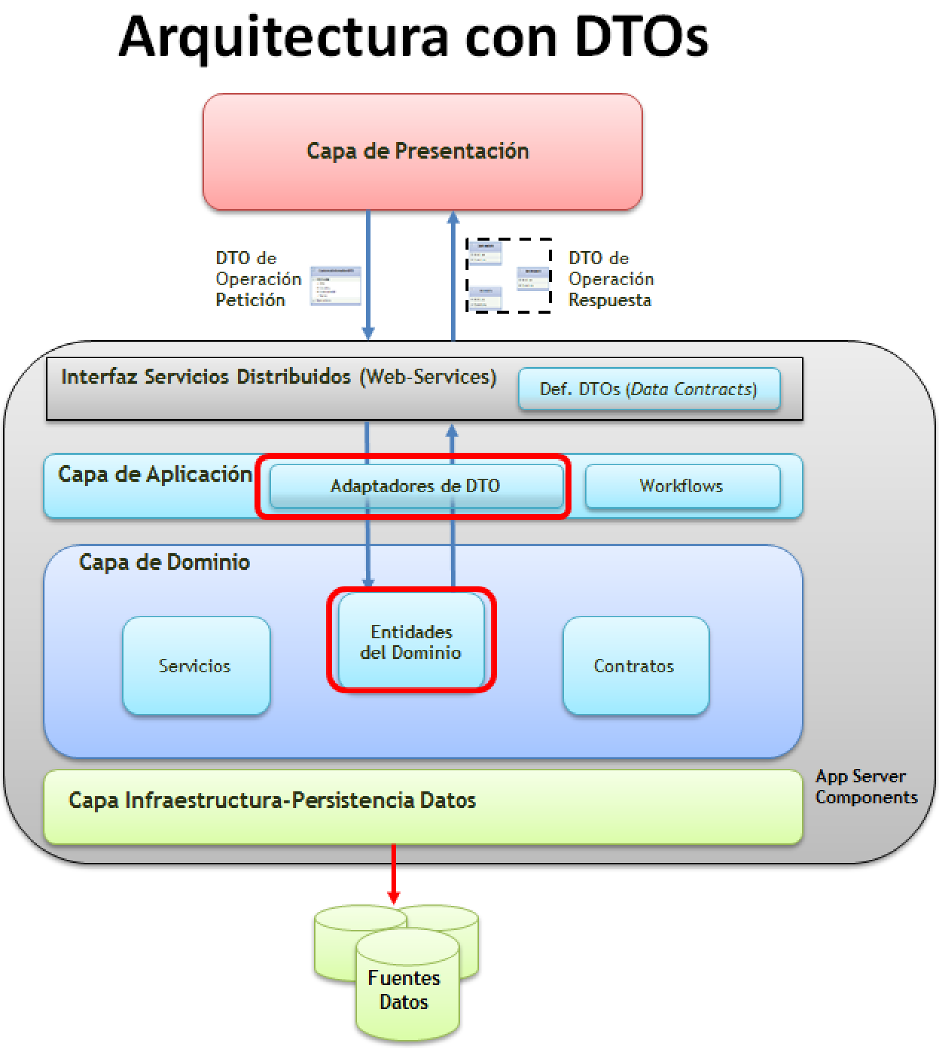


Figura 6.- Diagrama Arquitectura con DTOs (Data Transfer Objects)

En definitiva, se debe considerar la opción de hacer uso de **DTOs** (*Data Transfer Objects*), para consolidar datos en estructuras unificadas que minimicen el número de llamadas remotas a los Servicios Web. Los DTOs favorecen una granularización gruesa de operaciones al aceptar DTOs que están diseñados para transportar datos entre diferentes niveles físicos (Tiers).

Desde un punto de vista de Arquitectura de Software purista, este es el enfoque más correcto, pues desacoplamos las entidades de datos del Dominio de la forma en cómo se van a tratar los datos 'fuera del dominio' (Cliente u otra aplicación externa). A largo plazo, este desacoplamiento y uso de DTOs es el que más beneficios ofrece de cara a

cambios en uno u otro sito (Dominio vs. Capa de Presentación o consumidor externo). Sin embargo, el uso de DTOs requiere de un trabajo inicial bastante mayor que el hacer uso directamente de entidades del dominio serializadas (las cuales incluso pueden realizar tareas de control de concurrencia por nosotros), como veremos en el capítulo de implementación de Servicios en .NET. Supóngase un gran proyecto con cientos de entidades del dominio, el trabajo de creación de DTOs y adaptadores de DTOs crecerá exponencialmente. Debido a este sobre esfuerzo, también caben enfoques mixtos, por ejemplo, hacer uso de entidades del dominio serializadas para capas de presentación controladas, y uso de DTOs para una capa/fachada SOA hacia el exterior (otras consumidores desconocidos inicialmente).

Conjuntos de Registros/Cambios (*Artefactos* desconectados)

Los conjuntos de registros/cambios suelen ser implementaciones de datos compuestos desconectados, como pueda ser en .NET los *DataSets*. Suelen ser mecanismos muy fáciles de utilizar, pero están muy ligados a la tecnología subyacente, completamente acoplados a la tecnología de acceso a datos, por lo que son completamente contrarios al enfoque DDD (independencia de la capa de infraestructura) y en este tipo de Arquitectura orientada al dominio no serían recomendables. Probablemente si lo pueden ser en Arquitecturas para aplicaciones menos complejas y a desarrollar en un modo más RAD (*Rapid Application Development*).

Cualquiera de estos conceptos lógicos (Entidad, DTO, etc.) se podrá serializar a diferentes tipos de datos (XML, binario, diferentes formatos/esquemas XML, etc.) dependiendo de la implementación concreta elegida. Pero esta implementación tiene ya que ver con la tecnología, por lo que lo analizaremos en el capítulo de Implementación de Servicios Distribuidos en .NET, posteriormente.

Referencias sobre DTOs

Pros and Cons of Data Transfer Objects (Dino Esposito)
http://msdn.microsoft.com/en-us/magazine/ee236638.aspx

Building N-Tier Apps with EF4 (Danny Simons):
http://msdn.microsoft.com/en-us/magazine/ee335715.aspx

9.- CONSUMO DE SERVICIOS DISTRIBUIDOS BASADO EN AGENTES

Los Agentes de servicios básicamente establecen una sub-capa dentro de la aplicación cliente (Capa de Presentación) donde centralizar y localizar el ‘consumo’ de

Servicios-Web de una forma metódica y homogénea, en lugar de consumir directamente los Servicios desde cualquier parte de la aplicación cliente (formulario, página, etc.). El uso de agentes es en definitiva una forma (patrón) de diseñar y programar el consumo de Servicios Web.

Definición de Agente de Servicio

"*Un Agente de Servicio es un componente situado en la capa de presentación, y actua como front-end de comunicaciones hacia los Servicios-Web. Debe ser el único responsable de las acciones de consumo directo de Servicios-Web*".

Se podría definir también a un agente como una clase "*smart-proxy*" que sirve de intermediario entre un servicio y sus consumidores. Teniendo presente que el Agente se sitúa físicamente en el lado del cliente.

Desde el punto de vista de la aplicación cliente (WPF, Silverlight, OBA, etc.), un agente actúa 'en favor' de un Servicio-Web. Es decir, es como si fuera un 'espejo' local ofreciendo la misma funcionalidad que tiene el servicio en el servidor.

A continuación se muestra un esquema de los agentes situados en una arquitectura de 'consumo' de Servicios:

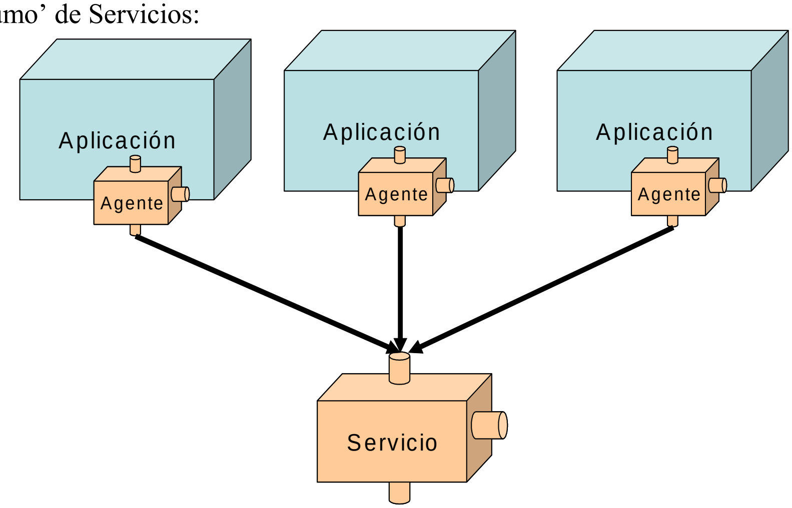


Figura 7.- Esquema de los agentes en una arquitectura de 'consumo' de Servicios

El **Agente** debe ayudar a preparar peticiones al servicio así como interpretar respuestas del servicio.

Es importante tener presente que un agente no es parte del servicio (debe mantener un desacoplamiento débil con el servicio) y por lo tanto el servicio no confía en el agente. Toda la interacción entre un agente y un servicio es siempre autenticada, autorizada y validada por el servicio de la misma forma que se accede a un servicio directamente sin un agente.

Algunas de las ventajas de hacer uso de Agentes, son:

- **Fácil integración**: Si un Servicio tiene ya desarrollado su correspondiente Agente, el proporcionar dicho agente ya desarrollado a quien va a consumir el servicio puede simplificar el proceso de desarrollo.

- **Gestión de errores**: Reaccionar correctamente a las condiciones de los errores es imprescindible y una de las tareas más complejas para el desarrollador que consume un servicio web desde una aplicación. Los Agentes deben de estar diseñados para entender los errores que pueda producir un servicio, simplificando en gran medida el desarrollo de integración posterior.

- **Gestión de datos off-line y cache**: Un agente puede estar diseñado para hacer 'cache' de datos del servicio de una forma correcta y entendible. Esto a veces puede mejorar espectacularmente los tiempos de respuesta (y por tanto el rendimiento y escalabilidad) de las peticiones e incluso permitir a las aplicaciones el trabajar de forma desconectada (*off-line)*.

- **Validación de peticiones**: Los agentes pueden comprobar los datos de entrada a enviar al servidor de componentes y asegurarse de que son correctos antes de realizar ninguna llamada remota (coste en tiempo de latencia al servidor). Esto no libera en absoluto al servidor de tener que validar los datos, pues en el servidor es la manera más segura (puede haber sido hacheado el cliente), pero si puede estar normalmente ahorrando tiempos.

- **Ruteo inteligente**: Algunos servicios pueden usar agentes para mandar peticiones a un servidor de servicio específico, basándose en el contenido de la petición.

En definitiva, el concepto es muy simple, los agentes son clases situadas en un *assembly* en el lado 'cliente' y son las únicas clases en el lado cliente que deberían interactuar con las clases proxy de los Servicios. A nivel práctico, lo normal es crearse proyectos de librerías de clases para implementar estas clases 'Agente'. Después, en la aplicación cliente simplemente tendremos que añadir una referencia a este *assembly*.

Antes de pasar a otros aspectos, quiero recalcar que el uso de Agentes es independiente de la tecnología, se puede hacer uso de este patrón consumiendo cualquier tipo de Servicio Distribuido.

10.- INTEROPERABILIDAD

Los principales factores que influencian la interoperabilidad de las aplicaciones son la disponibilidad de canales de comunicación apropiados (estándar) así como los formatos y protocolos que pueden entender los participantes de diferentes tecnologías.Considerad las siguientes guías:

- Para conseguir una comunicación con una variedad de plataformas y dispositivos de diferentes fabricantes, es recomendable hacer uso de protocolos y formatos de datos estándar, como HTTP y XML, respectivamente. Hay que tener en cuenta que las decisiones sobre protocolo pueden afectar a la disponibilidad de clientes objetivo disponibles. Por ejemplo, los sistemas objetivos podrían estar protegidos por 'Firewalls' que bloqueen algunos protocolos.

- El formato de datos elegido puede afectar a la interoperabilidad. Por ejemplo, los sistemas objetivo pueden no entender tipos específicos ligados a una tecnología (problema existente por ejemplo con Datasets de ADO.NET hacia sistemas JAVA), o pueden tener diferentes formas de gestionar y serializar los tipos de datos.

- Las decisiones de cifrado y descifrado de las comunicaciones pueden afectar también a la interoperabilidad. Por ejemplo, algunas técnicas de cifrado/descifrado de mensajes pueden no estar disponibles en todos los sistemas.

11.- RENDIMIENTO

El diseño de los interfaces de comunicación y los formatos de datos que se utilicen tendrán un impacto considerable en el rendimiento de la aplicación, especialmente cuando cruzamos 'fronteras' en la comunicación entre diferentes procesos y/o diferentes máquinas. Mientras otras consideraciones, como interoperabilidad, pueden requerir interfaces y formatos de datos específicos, hay técnicas que podemos utilizar para mejorar el rendimiento relacionado con las comunicaciones entre diferentes niveles (Tiers) de la aplicación.

Considerad las siguientes guías y mejores prácticas:

- Minimizar el volumen de datos trasmitidos por la red, esto reduce sobrecargas de serialización de objetos.

- Es muy importante a tener en cuenta que se debe siempre evitar trabajar con interfaces de Servicios Web con una fina granularización (que es como internamente están diseñados normalmente los componentes internos del Dominio). Esto es problemático porque obliga a implementar el consumo de Servicios-web en un modo muy de tipo 'conversación'). Ese tipo de diseño impacta fuertemente en el rendimiento pues obliga a la aplicación cliente a realizar muchas llamadas remotas para una única unidad de trabajo y puesto que las invocaciones remotas tienen un coste en rendimiento (activación de Servicio-Web, serialización/des-serialización de datos, etc.), es crítico minimizar el número de llamadas. Para esto es útil el uso de DTOs cuando se identifique conveniente (Permite agrupar diferentes entidades del Dominio en

una única estructura de datos a transferir), si bien también hay nuevas tecnologías ORM (como 'Entity Framework') que permiten serializar grafos que incluyan varias.

- Considerar el uso de una Fachada de Servicios Web que proporcionen una granularización más gruesa de los interfaces, envolviendo a los componentes de negocio del Dominio que normalmente si dispondrán de una 'granularización fina'.

- Si el rendimiento en la serialización es crítico para el rendimiento de la aplicación, considerar el uso de clases propias con serialización binaria (si bien, la serialización binaria normalmente no será interoperable con plataformas tecnológicas diferentes).

- El uso de otros protocolos diferentes a HTTP (como TCP, Named-Pipes, MSMQ, etc.) también pueden mejorar sustancialmente, en ciertos casos, el rendimiento de las comunicaciones. Sin embargo, podemos perder la interoperabilidad de HTTP.

12.- COMUNICACIÓN ASÍNCRONA VS. SÍNCRONA

Debemos tener en cuenta las ventajas e inconvenientes de realizar la comunicación de Servicios-web de una forma síncrona versus asíncrona.

La comunicación **síncrona** es más apropiada para escenarios donde debemos garantizar el orden en el cual se realizan las llamadas o cuando el usuario debe esperar a que la llamada devuelva resultados (si bien esto se puede conseguir también con comunicación asíncrona).

La comunicación **asíncrona** es más apropiada para escenarios en los que la inmediatez en la respuesta de la aplicación debe ser inmediata o en escenarios donde no se puede garantizar que el objetivo esté disponible.

Considerad las siguientes guías a la hora de decidir por implementaciones síncronas o asíncronas:

- Para conseguir un máximo rendimiento, bajo acoplamiento y minimizar la carga del sistema, se debe considerar el modelo de **comunicación asíncrona**. Si algunos clientes solo pueden realizar llamadas síncronas, se puede implementar un componente (Agente de Servicio en el cliente) que hacia el cliente sea síncrono, pero en cambio él mismo consuma los servicios web de forma asíncrona, lo cual da la posibilidad de realizar diferentes llamadas en paralelo, aumentando el rendimiento global en esa área.

- En casos en los que se debe garantizar el orden en el que las operaciones se ejecutan o donde se hace uso de operaciones que dependen del resultado de operaciones previas, probablemente el esquema más adecuado sea una

comunicación síncrona. La mayoría de las veces un funcionamiento síncrono con cierto orden, se puede simular con operaciones asíncronas coordinadas, sin embargo, dependiendo del escenario concreto, el esfuerzo en implementarlo puede o no merecer la pena.

- Si se elige el uso de comunicación asíncrona pero no se puede garantizar que exista siempre una conectividad de red y/o disponibilidad del destino, considerar hacer uso de un sistema de mensajería 'guardar/enviar' **que aseguran la comunicación** (Sistema de **Colas de Mensajes**, como puede ser MSMQ), para evitar la pérdida de mensajes. Estos sistema avanzados de colas de mensajes pueden incluso extender transacciones con el envío de mensajes asíncronos a dichas colas de mensajes. Si adicionalmente se necesita interoperar e integrar con otras plataformas empresariales, considerar el uso de plataformas de integración (como Microsoft Biztalk Server).

13.- REST VS. SOAP

REST (*Representational State Transfer*) y SOAP (*Simple Object Access Protocol*) **representan dos estilos** bastante diferentes para implementar una Arquitectura de Servicios Distribuidos. Técnicamente, REST es un patrón de arquitectura construido con verbos simples que encajan perfectamente con HTTP. Si bien, aunque los principios de Arquitectura de REST podrían aplicarse a otros protocolos adicionales a HTTP, en la práctica, las implementaciones de REST se basan completamente en HTTP.

SOAP es un protocolo de mensajería basado en XML (mensajes SOAP con un formato XML concreto) que puede utilizarse con cualquier protocolo de comunicaciones (Transporte), incluido el propio HTTP.

La principal diferencia entre estos dos enfoques es la forma en la que se mantiene el estado del servicio. Y nos referimos a un estado muy diferente al de estado de sesión o aplicación. Nos referimos a los diferentes estados por los que una aplicación pasa durante su vida. Con SOAP, el movimiento por los diferentes estados se realiza interactuando con un extremo único del servicio que puede encapsular y proporcionar acceso a muchas operaciones y tipos de mensaje.

Por el contrario, con REST, se permite un número limitado de operaciones y dichas operaciones se aplican a recursos representados y direccionados por URIs (direcciones HTTP). Los mensajes capturan el estado actual o el requerido del recurso. REST funciona muy bien con aplicaciones Web donde se puede hacer uso de HTTP para tipos de datos que no son XML. Los consumidores del servicio interactúan con los recursos mediante URIs de la misma forma que las personas pueden navegar e interactuar con páginas Web mediante URLs (direcciones web).

La siguiente figura trata de mostrar en qué escenarios REST o SOAP encajan mejor:

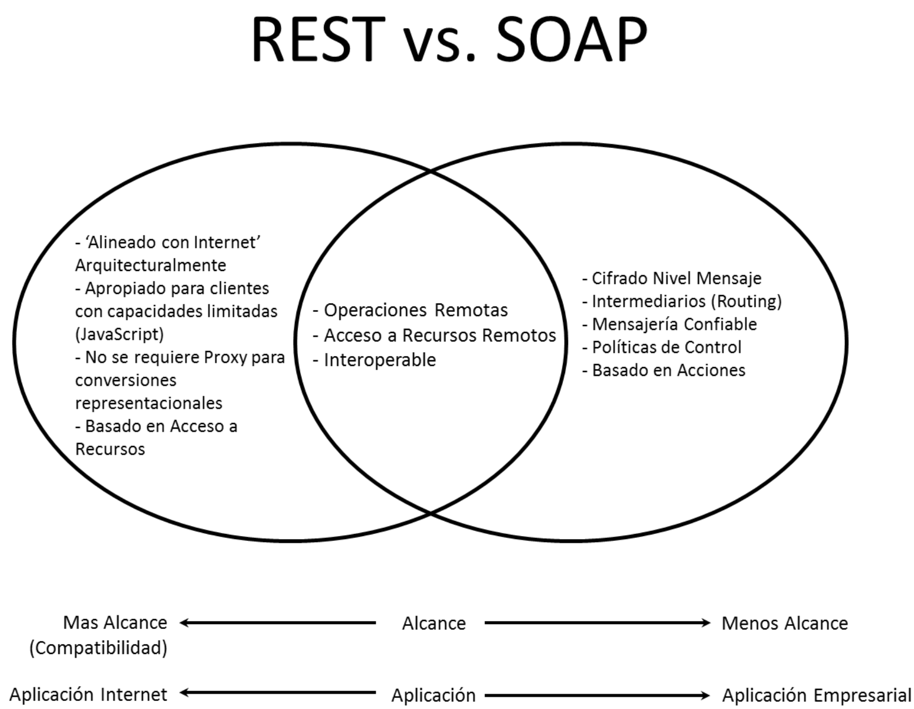


Figura 8.- Escenarios REST o SOAP

Desde un punto de vista tecnológico, estas son algunas ventajas y desventajas de ambos:

Ventajas de SOAP:

- Bueno para datos (Las comunicaciones son estrictas y estructuradas)

- Dispone de proxies fuertemente tipados gracias a WSDL

- Funciona sobre diferentes protocolos de comunicaciones. El uso de otros protocolos no HTTP (como TCP, NamedPipes, MSMQ, etc.), puede mejorar rendimiento en ciertos escenarios.

Desventajas de SOAP:

- Los mensajes de SOAP no son 'cacheables'

- No se puede hacer uso de mensajes SOAP en JavaScript (Para AJAX debe utilizarse REST).

Ventajas de REST:

- Gobernado por las especificaciones HTTP por lo que los servicios actúan como Recursos, igual que Imágenes o documentos HTML.
- Los Datos pueden bien mantenerse de forma estricta o de forma desacoplada (no tan estricto como SOAP)
- Los Recursos REST pueden consumirse fácilmente desde código JavaScript (AJAX, etc.)
- Los mensajes son ligeros, por lo que el rendimiento y la escalabilidad que ofrece es muy alta. Importante para Internet.
- REST puede utilizar bien XML o JSON como formato de los datos.

Desventajas de REST:

- Es difícil trabajar con objetos fuertemente tipados en el código del servidor, aunque esto depende de las implementaciones tecnológicas y está mejorando en las últimas versiones.
- Solo funciona normalmente sobre HTTP (No es bueno apra aplicaciones de alto rendimiento y tiempo real, como aplicaciones de bolsa, etc.)
- Las llamadas a REST están restringidas a Verbos HTTP (GET, POST, PUT, DELETE, etc.)

Aun cuando ambas aproximaciones (REST y SOAP) pueden utilizarse para diferentes tipos de servicios, la aproximación basada en REST es normalmente más adecuada para Servicios Distribuidos accesibles públicamente (Internet) o en casos en los que un Servicio puede ser accedido por consumidores desconocidos. SOAP se ajusta, por el contrario, mucho mejor a implementar rangos de implementaciones procedurales, como un interfaz entre las diferentes Capas de una Arquitectura de aplicación o en definitiva, Aplicaciones Empresariales privadas.

Con SOAP no estamos restringidos a HTTP. Las especificaciones estándar WS-*, que pueden utilizarse sobre SOAP, proporcionan un estándar y por lo tanto un camino interoperable para trabajar con aspectos avanzados como SEGURIDAD, TRANSACCIONES, DIRECCIONAMIENTO y FIABILIDAD.

REST ofrece también por supuesto un gran nivel de interoperabilidad (debido a la sencillez de su protocolo), sin embargo, para aspectos avanzados como los anteriormente mencionados, sería necesario implementar mecanismos propios que dejarían de ser un estándar.

En definitiva, ambos protocolos permiten intercambiar datos haciendo uso de verbos, la diferencia está en que con REST, ese conjunto de verbos está restringido a la coincidencia con los verbos HTTP (GET, PUT, etc.) y en el caso de SOAP, el conjunto de verbos es abierto y está definido en el *endpoint* del Servicio.

Considerad las siguientes guías a la hora de elegir por una u otra aproximación:

- SOAP es un protocolo que proporciona un marco base de mensajería sobre el que se puede construir una abstracción de capas y se utiliza normalmente como un sistema de llamadas tipo RPC (bien síncrono o asíncrono) que pasa llamadas y respuestas y cuyos datos comunicados por la red es en la forma de mensajes XML.

- SOAP gestiona aspectos como seguridad y direccionamiento mediante su implementación interna del protocolo SOAP.

- REST es una técnica que utiliza otros protocolos, como JSON (JavaScript Object Notation), Atom como protocolo de publicación, y formatos sencillos y ligeros tipo POX (Plain Old XML).

- REST permite hacer uso de llamadas estándar HTTP como GET y PUT para realizar consultas y modificar el estado del sistema. REST es *stateless* por naturaleza, lo cual significa que cada petición individual enviada desde el cliente al servidor debe contener toda la información necesaria para entender la petición puesto que el servidor no almacenará datos de estado de sesión.

13.1.-Consideraciones de Diseño para SOAP

SOAP es un protocolo basado en mensajes que se utiliza para implementar la capa de mensajes de un Servicio-Web. El mensaje está compuesto por un 'sobre' que contiene una cabecera (header) y un cuerpo (body). La cabecera puede utilizarse para proporcionar información que es externa a la operación a ser realizada por el servicio (p.e. aspectos de seguridad, transaccionales o enrutamiento de mensajes, incluidos en la cabecera).

El 'cuerpo' del mensaje contiene contratos, en forma de esquemas XML, los cuales se utilizan para implementar el servicio web. Considerar las siguientes guías de diseño especificas para Servicios-Web SOAP:

- Determinar cómo gestionar errores y faltas (normalmente excepciones originadas en capas internas del servidor) y como devolver información apropiada de errores al consumidor del Servicio Web. (Ver "ExceptionHandling in Service Oriented Applications" en **http://msdn.microsoft.com/en-us/library/cc304819.aspx**).

- Definir los esquemas de las operaciones que puede realizar un servicio (Contrato de Servicio), las estructuras de datos pasadas cuando se realizan peticiones (Contrato de datos) y los errores o faltas que pueden devolverse desde una petición del servicio web.

- Elegir un modelo de seguridad apropiado. Para más información ver "*Improving Web Services Security*: Scenarios and Implementation Guidance for WCF" en **http://msdn.microsoft.com/en-us/library/cc949034.aspx**

- Evitar el uso de tipos complejos y dinámicos en los esquemas (como Datasets). Intentar hacer uso de tipos simples o clases DTO o entidad simples para maximizar la interoperabilidad con cualquier plataforma.

13.2.-Consideraciones de Diseño para REST

REST representa un estilo de arquitectura para sistemas distribuidos y está diseñado para reducir la complejidad dividiendo el sistema en recursos. Los recursos y las operaciones soportadas por un recurso se representan y exponen mediante un conjunto de URIs (direcciones HTTP) lógicamente sobre protocolo HTTP. Considerar las siguientes guías específicas para REST:

- Identificar y categorizar los recursos que estarán disponibles para los consumidores del Servicio

- Elegir una aproximación para la representación de los recursos. Una buena práctica podría ser el hacer uso de nombres con significado (¿Lenguaje Ubicuo en DDD?) para los puntos de entrada REST e identificadores únicos para instancias de recursos específicos. Por ejemplo, **http://www.miempresa.empleado/** representa el punto de entrada para acceder a un empleado y **http://www.miempresa.empleado/perez01** utiliza un ID de empleado para indicar un empleado específico.

- Decidir si se deben soportar múltiples representaciones para diferentes recursos. Por ejemplo, podemos decidir si el recurso debe soportar un formato XML, Atom o JSON y hacerlo parte de la petición del recurso. Un recurso puede ser expuesto por múltiples representaciones (por ejemplo, **http://www.miempresa.empleado/perez01.atom** y **http://www.miempresa.empleado/perez01.json**).

- Decidir si se van a soportar múltiples vistas para diferentes recursos. Por ejemplo, decidir si el recurso debe soportar operaciones GET y POST o solo operaciones GET. Evitar el uso excesivo de operaciones POST, si es posible, y evitar también exponer acciones en la URI.

- No implementar el mantenimiento de estados de sesión de usuario dentro de un servicio y tampoco intentar hacer uso de HIPERTEXTO (como los controles escondidos en páginas Web) para gestionar estados. Por ejemplo, cuando un usuario realiza una petición como añadir un elemento al carro de la compra de un Comercio-e, los datos del carro de la compra deben almacenarse en un

almacén persistente de estados o bien un sistema de cache preparado para ello, pero no como estados de los propios servicios (lo cual además invalidaría escenarios escalables tipo 'Web-Farm').

14.- INTRODUCCIÓN A SOAP Y WS-*

SOAP, originalmente definido como 'Simple Object Access Protocol', es una especificación de para intercambio de información estructurada en la implementación de Servicios Web. Se basa especialmente en XML como formato de mensajes y HTTP como protocolo de comunicaciones.

SOAP es la base del stack de protocolos de Servicios web, proporcionando un marco base de mensajería sobre el cual se pueden construir los Servicios Web.

Este protocolo está definido en tres partes:

- Un sobre de mensaje, que define qué habrá en el 'cuerpo' o contenido del mensaje y como procesarlo
- Un conjunto de reglas de serialización para expresar instancias de tipos de datos de aplicación
- Una convención para representar llamadas y respuestas a métodos remotos

En definitiva, es un sistema de invocaciones remotas basado a bajo nivel en mensajes XML. Un mensaje SOAP se utilizará tanto para solicitar una ejecución de un método de un Servicio Web remoto así como se hará uso de otro mensaje SOAP como respuesta (conteniendo la información solicitada). Debido a que el formato de los datos es XML (texto, con un esquema, pero texto, al fin y al cabo), puede llegar a integrarse en cualquier plataforma tecnológica. SOAP es interoperable.

El estándar básico de SOAP es 'SOAP WS-I *Basic Profile'*.

15.- ESPECIFICACIONES WS-*

Servicios Web básicos (como SOAP WS-I, *Basic Profile*) nos ofrecen poco más que comunicaciones entre el servicio Web y las aplicaciones cliente que lo consumen. Sin embargo, los estándares de servicios web básicos (*WS-Basic Profile*), fueron simplemente el inicio de SOAP.

Las aplicaciones empresariales complejas y transaccionales requieren de muchas más funcionalidades y requisitos de calidad de servicio (QoS) que simplemente una comunicación entre cliente y servicio web. Los puntos o necesidades más destacables de las aplicaciones empresariales pueden ser aspectos como:

- Seguridad avanzada orientada a mensajes en las comunicaciones con los servicios, incluyendo diferentes tipos de autenticación, autorización, cifrado, no tampering, firma, etc.
- Mensajería estable y confiable
- Soporte de transacciones distribuidas entre diferentes servicios.
- Mecanismos de direccionamiento y ruteo.
- Metadatos para definir requerimientos como políticas.
- Soporte para adjuntar grandes cantidades de datos binarios en llamadas/respuestas a servicios (imágenes y/o 'attachments' de cualquier tipo).

Para definir todas estas 'necesidades avanzadas', la industria (diferentes empresas como Microsoft, IBM, HP, Fujitsu, BEA, VeriSign, SUN, Oracle, CA, Nokia, CommerceOne, Documentum, TIBCO, etc.) han estado y continúan definiendo unas especificaciones teóricas que conformen como deben funcionar los 'aspectos extendidos' de los Servicios Web.

A todas estas 'especificaciones teóricas' se las conoce como las **especificaciones WS.***. (El '*' viene dado porque son muchas especificaciones de servicios web avanzados, como *WS-Security*, *WS-SecureConversation*, WS-AtomicTransactions, etc.). Si se quiere conocer en detalle estas especificaciones, se pueden consultar los estándares en: **http://www.oasis-open.org**.

En definitiva, esas especificaciones WS-* definen teóricamente los requerimientos avanzados de las aplicaciones empresariales.

El siguiente esquema muestra a alto nivel las diferentes funcionalidades que trata de resolver WS.*.

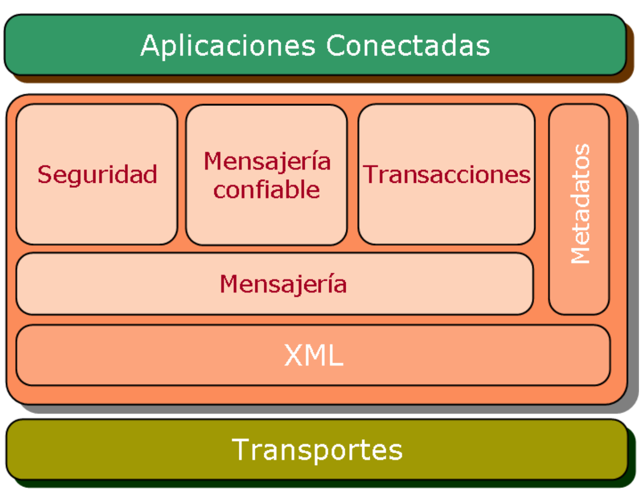


Figura 9.- Esquema con las diferentes funcionalidades para resolver WS.*.

Todos los módulos centrales (Seguridad, Mensajería Confiable, Transacciones y Metadatos) son precisamente las funcionalidades de las que carecen los Servicios Web XML básicos y lo que define precisamente WS.*.

Las **especificaciones WS.*** están así mismo formadas por subconjuntos de especificaciones:

- WS-Security
- WS-Messaging
- WS-Transaction
- WS-Reliability
- WS-Metadata

Estos a su vez se vuelven a dividir en otros subconjuntos de especificaciones aún más definidas, como muestra el siguiente cuadro:

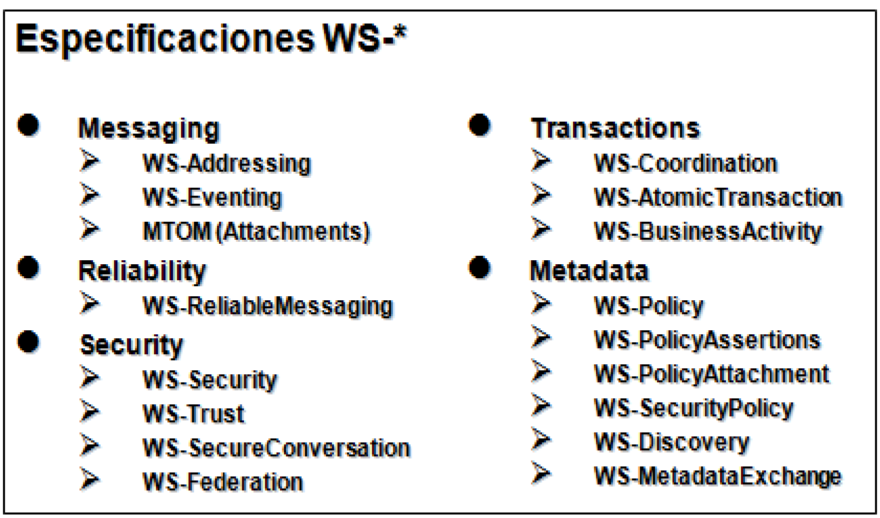


Figura 10.- Especificaciones WS-*

Como se puede suponer, las especificaciones WS-* son prácticamente todo un mundo, no es algo pequeño y limitado que simplemente 'extienda un poco' a los Servicios Web básicos.

A continuación mostramos una tabla donde podemos ver las necesidades existentes en las aplicaciones SOA distribuidas empresariales y los estándares WS-* que los definen:

Tabla 2.- Especificaciones WS-*

Necesidades avanzadas en los Servicios	Especificaciones WS-* que lo definen
• Seguridad avanzada orientada a mensajes en las comunicaciones con los servicios, incluyendo diferentes tipos de autenticación, autorización, cifrado, no tampering, firma, etc.	- WS-Security - WS-SecureConversation - WS-Trust
• Mensajería estable y confinable	- WS-ReliableMessaging
• Soporte de transacciones distribuidas entre diferentes servicios	- WS-AtomicTransactions
• Mecanismos de direccionamiento y ruteo	- WS-Addressing
• Metadatos para definir requerimientos como políticas	- WS-Policy
• Soporte para adjuntar grandes cantidades de datos binarios en llamadas/respuestas a servicios (imágenes y/o 'attachments' de cualquier tipo)	- MTOM

16.- INTRODUCCIÓN A REST

¿Qué es REST?, bien, REST fue introducido por *Roy Fielding* en una ponencia, donde describió un "estilo de arquitectura" de sistemas interconectados. Así mismo, REST es un acrónimo que significa "*Representational State Transfer*".

¿Por qué se le llama "*Representational State Transfer*"?. Pues, en definitiva, el Web es un conjunto de recursos. Un recurso es un elemento de interés. Por ejemplo, Microsoft puede definir un tipo de recurso sobre un producto suyo, que podría ser, Microsoft Office SharePoint. En este caso, los clientes pueden acceder a dicho recurso con una URL como:

http://www.microsoft.com/architecture/guides

Para acceder a dicho recurso, se devolverá una **representación** de dicho recurso (p.e. ArchitectureGuide.htm). La **representación** sitúa a la aplicación cliente en un **estado**. El resultado del cliente accediendo a un enlace dentro de dicha página HTML será otro recurso accedido. La nueva representación situará a la aplicación cliente en otro estado. Así pues, la aplicación cliente cambia (**transfiere**) de estado con cada representación de recurso. En definitiva "*Representational State Transfer*".

En definitiva, los objetivos de REST son plasmar las características naturales del Web que han hecho que Internet sea un éxito. Son precisamente esas características las que se han utilizado para definir REST.

REST no es un estándar, es un estilo de arquitectura. No creo que veamos a W3C publicando una especificación REST, porque REST es solamente un estilo de arquitectura. No se puede "empaquetar" un estilo, solo se puede comprender y diseñar los servicios web según ese estilo. Es algo comparable a un estilo de arquitectura "N-Tier" o arquitectura SOA, no hay un estándar N-Tier ni estándar SOA.

Sin embargo, aunque REST no es un estándar en sí mismo, sí que está basado en estándares de Internet:

- HTTP
- URL
- XML/HTML/PNG/GIF/JPEG/etc (Representaciones de recursos)
- Text/xml, text/html, image/gif, etc. (tipos MIME)

16.1.-La URI en REST

En definitiva y como concepto fundamental en REST, lo más importante en REST es la URI (URI es una forma más *cool* y técnica de decir URL, por lo que es mejor es

mejor decirlo así…). Al margen de bromas, la URI es muy importante en REST porque basa todas las definiciones de acceso a Servicios Web en la sintaxis de una URI. Para que se entienda, vamos a poner varios ejemplos de URIs de Servicios Web basados en REST. Como se verá, es auto explicativo según su definición, y ese es uno de los objetivos de REST, simplicidad y auto explicativo, por lo que ni voy a explicar dichas URIs ejemplo:

http://www.midominio.com/Proveedores/ObtenerTodos/
http://www.midominio.com/Proveedores/ObtenerProveedor/2050
http://www.midominio.com/Proveedores/ObtenerProveedorPorNombre/Ramirez/Jose

Me reafirmo, y lo dejamos sin explicar por lo entendibles que son.

16.2.-Simplicidad

La simplicidad es uno de los aspectos fundamentales en REST. Se persigue la simplicidad en todo, desde la URI hasta en los mensajes XML proporcionados o recibidos desde el servicio Web. Esta simplicidad es una gran diferencia si lo comparamos con SOAP, que puede tener gran complejidad en sus cabeceras, etc.

El beneficio de dicha simplicidad es poder conseguir un buen rendimiento y eficiencia (aun cuando estemos trabajando con estándares no muy eficientes, como HTTP y XML), pero al final, los datos (bits) que se transmiten son siempre los mínimos necesarios, tenemos algo ligero, por lo que el rendimiento será bastante óptimo. Por el contrario, como contrapartida tendremos que si nos basamos en algo bastante simple, habrá muchas cosas complejas que si se pueden implementar con estándares SOAP, que con REST es imposible, por ejemplo, estándares de seguridad, firma y cifrado a nivel de mensajes, transaccionalidad que englobe diferentes servicios web y un largo etc. de funcionalidades avanzadas que si se definen en las especificaciones WS-* basado en SOAP.

Pero el objetivo de REST no es conseguir una gran o compleja funcionalidad, es conseguir un mínimo de funcionalidad necesaria en un gran porcentaje de servicios web en Internet, que sean interoperables, que simplemente se transmita la información y que sean servicios Web muy eficientes.

A continuación muestro un ejemplo de un mensaje REST devuelto por un Servicio Web. Contrasta su simplicidad con un mensaje SOAP WS-* que puede llegar a ser muy complejo y por lo tanto también más pesado. Los mensajes REST son muy ligeros:

```
<?xml version="1.0"?>
<p:Cliente xmlns:p="http://www.miempresa.com"
        xmlns:xlink="http://www.w3.org/1999/xlink">
      <Cliente-ID>00345</Cliente-ID>
      <Nombre>Ramirez y asociados</Nombre>
```

```
            <Descripcion>Empresa dedicada a automoción</ Descripcion >
            <Detalles                  xlink:href="http://www.miempresa.com
    /clientes/00345/detalles"/>
       </p:Cliente>
```

Como se puede observar, es difícil diseñar un mensaje XML más simplificado que el anterior. Es interesante destacar el elemento "Detalles" del ejemplo, que es de tipo enlace o hiperenlace. Más adelante se resalta la importancia de estos elementos de tipo "enlace".

16.3.-URLs lógicas versus URLs físicas

Un recurso es una entidad conceptual. Una representación es una manifestación concreta de dicho recurso. Por ejemplo:

http://www.miempresa.com/clientes/00345

La anterior URL es una URL lógica, no es una URL física. Por ejemplo, no hace falta que exista una página HTML para cada cliente de este ejemplo.

Un aspecto de diseño correcto de URIs en REST es que no se debe revelar la tecnología utilizada en la URI/URL. Se debe poder tener la libertad de cambiar la implementación sin impactar en las aplicaciones cliente que lo están consumiendo. De hecho, esto supone un problema para servicios WCF albergados en IIS, puesto que dichos servicios funcionan basados en una página .svc. Pero existen trucos para ocultar dichas extensiones de fichero.

16.4.-Características base de Servicios Web REST

- Cliente-Servidor: Estilo de interacción "pull". Métodos complejos de comunicaciones tipo Full-Duplex o Peer-to-Peer quedan lejos de poder ser implementadas con REST. REST es para Servicios Web sencillos.

- *Stateless* (Sin estados): Cada petición que hace el cliente al servidor debe contener toda la información necesaria para entender y ejecutar la petición. No puede hacer uso de ningún tipo de contexto del servidor. Así es como son también los servicios Web básicos en .NET (single-call, sin estados), sin embargo, en WCF existen más tipos de instanciación como Singleton y shared (con sesiones). Esto también queda lejos de poder implementarse con un Servicio Web REST.

- *Cache*: Para mejorar la eficiencia de la red, las respuestas deben poder ser clasificadas como "*cacheables*" y "no *cacheables*"

- Interfaz uniforme: Todos los recursos son accedidos con un interfaz genérico (ej: HTTP GET, POST, PUT, DELETE), sin embargo, el interfaz mas importante o preponderante en REST es GET (como las URLs mostradas de ejemplo anteriormente). GET es considerado "especial" para REST.

- El tipo de contenido (content-type) es el modelo de objetos

- Imagen, XML, JSON, etc.

- Recursos nombrados. El sistema está limitado a recursos que puedan ser nombrados mediante una URI/URL.

- Representaciones de recursos interconectados: Las representaciones de los recursos se interconectan mediante URLs, esto permite a un cliente el progresar de un estado al siguiente.

16.5.-Principios de Diseño de Servicios Web REST

1. La clave para crear servicios web en una red REST (p.e. el Web en Internet) es identificar todas las entidades conceptuales que se desea exponer como servicios. Antes vimos algunos ejemplos, como clientes, o productos, facturas, etc.

2. Crear una URI/URL para cada recurso. Los recursos deben ser nombres, no verbos. Por ejemplo, no se debe utilizar esto:

   http://www.miempresa.com/clientes/obtenercliente?id=00452

   Estaría incorrecto el verbo obtenercliente. En lugar de eso, se pondría solo el nombre, así:

   http://www.miempresa.com/clientes/clientes/00452

3. Categorizar los recursos de acuerdo a si las aplicaciones cliente pueden recibir una representación del recurso, o si las aplicaciones cliente pueden modificar (añadir) al recurso. Para lo primero, se debe poder hacer accesible el recurso con un HTTP GET, para lo segundo, hacer accesible los recursos con HTTP POST, PUT y/o DELETE.

4. Las representaciones no deben ser islas aisladas de información. Por eso se debe implementar enlaces dentro de los recursos para permitir a las aplicaciones cliente el consultar más información detallada o información relacionada.

5. Diseñar para revelar datos de forma gradual. No revelar todo en una única respuesta de documento. Proporcionar enlaces para obtener más detalles.

6. Especificar el formato de la respuesta utilizando un esquema XML (W3C Schema, etc.)

Recursos adicionales

- "Enterprise Solution Patterns Using Microsoft .NET" en **http://msdn.microsoft.com/en-us/library/ms998469.aspx**
- "Web Service Security Guidance" en **http://msdn.microsoft.com/en-us/library/aa480545.aspx**
- "Improving Web Services Security: Scenarios and Implementation Guidance for WCF" en **http://www.codeplex.com/WCFSecurityGuide**
- "WS-* Specifications" en **http://www.ws-standards.com/ws-atomictransaction.asp**
-

17.- ODATA: OPEN DATA PROTOCOL

OData es un concepto de más alto nivel que SOAP y REST. También es más novedoso en el tiempo, pues es una propuesta de estándar de protocolo de alto nivel basado en **REST** y **AtomPub**.

Empecemos por el principio. **¿Qué es entonces exactamente OData?**.

Tabla 3.- Definición de ODAta

Definición	
	OData (*Open Data Protocol*) es un protocolo web para realizar consultas y actualizaciones remotas para acceder a servicios y almacenes de datos. OData emergió basándose en las experiencias de implementaciones de servidores y clientes **AtomPub**. OData se utiliza para exponer y acceder a información de diferentes recursos, incluyendo pero no limitándose a bases de datos relacionales. Realmente puede publicar cualquier tipo de recursos.

OData está basado en algunas convenciones, especialmente sobre **AtomPub** haciendo uso de servicios **REST** con orientación a datos. Estos servicios comparten recursos identificados mediante el uso de URIs (*Uniform Resource Identifiers*) y definidos como un modelo abstracto de datos para ser leídos/consultados y editados por clientes de dichos servicios web HTTP-REST.

OData está formado por un conjunto de especificaciones como [OData:URI], [OData:Terms], [OData:Operations], [OData:Atom], [OData:JSON] y [OData:Batch].

Así pues, OData es un **protocolo de alto nivel diseñado para compartir datos 'en la red'**, especialmente en entornos públicos e interoperables de Internet. En definitiva, es un nivel más arriba de REST, pero siguiendo su misma tendencia, utilizando URIs para identificar cada pieza de información en un servicio, HTTP para transportar peticiones y respuestas y AtomPub y JSON para manejar colecciones y representaciones de datos.

El objetivo principal de OData es ofrecer una forma estándar de consumir datos a través de la red y conseguir que los consumidores de los servicios de datos hagan uso de una serie de convenciones de alto nivel que tendrán muchísimo interés si se adoptan mayoritariamente. En definitiva es hacer uso de esquemas y convenciones predefinidas en lugar de 'reinventar la rueda' durante el desarrollo y nacimiento de cada servicio distribuido o servicio web.

Finalmente, recalcar que OData es un estándar propuesto por Microsoft que nace inicialmente de los protocolos utilizados originalmente en ***ADO.NET Data Services*** (actualmente llamado ***WCF Data Services***), pero lo interesante es que Microsoft lo ha evolucionado y liberado mediante el **OSP** (*Open Specification Promise*) para que cualquier fabricante pueda crear implementaciones de OData.

Los beneficios de OData como protocolo abierto y estándar propuesto son la interoperabilidad y colaboración con otras plataformas y la forma en la que se puede implementar por cualquier plataforma que soporte HTTP, XML, AtomPub y JSON. Por ejemplo, **IBM Web Sphere** es uno de los productos y fabricantes que soporta OData (el servicio *IBM WebSphere eXtreme Scale REST* soporta OData), junto con bastantes tecnologías y productos de Microsoft, fundamentalmente los siguientes:

- Tecnología/implementación base de OData de Microsoft
  - ***WCF Data Services***
- Productos y tecnologías de nivel superior:
  - *Windows Azure Storage Tables*
  - *SQL Azure*
  - *SharePoint 2010*
  - *SQL Server Reporting Services*
  - *Excel 2010 (con SQL Server PowerPivot for Excel)*

Para conocer la lista complete, consultar **http://www.odata.org/producers**

Debido a su naturaleza (REST, Web, AtomPub e interoperabildiad) está bastante más orientado a publicación y consumo de datos en entornos heterogéneos e Internet y por lo tanto, servicios más bien '*Data Oriented*' y no tanto '*Domain Oriented*' (DDD). Probablemente en una aplicación empresarial compleja y privada, la implementación de sus servicios distribuidos internos sea más potente el uso de SOAP y especificaciones WS-* (Seguridad, transacciones, etc.). Sin embargo, también es posible que una aplicación '*Domain Oriented*' quiera publicar información hacia el exterior (otras aplicaciones y o servicios inicialmente desconocidos). Ahí es donde OData encajaría perfectamente como un interfaz de acceso adicional a nuestra aplicación/servicio '*Domain Oriented*' desde el mundo exterior, otros servicios y en definitiva 'la red'.

Actualmente, en nuestra implementación de la Arquitectura presente (*NCapas con Orientación al Dominio*) no hacemos uso de OData porque DDD no ofrece una arquitectura/aplicación '*Data Oriented*', es '*Domain Oriented*' y los servicios distribuidos los estamos consumiendo fundamentalmente desde otra capa de nuestra aplicación (Capa de Presentación <u>dentro de nuestra aplicación</u>), por lo que es más flexible hacer uso de SOAP o incluso REST a más bajo nivel. OData está más orientado a publicar directamente datos en forma de servicios CRUD (*Create-Read-Update-Delete*), con unas especificaciones pre-establecidas, hacia donde está orientado *WCF Data Services*.

18.- REGLAS GLOBALES DE DISEÑO PARA SISTEMAS Y SERVICIOS SOA

Tabla 4.- Reglas globales de diseño

Regla Nº: D22	**Identificar qué componentes de servidor deben ser Servicios-SOA**

- **<u>Norma</u>**
- No todos los componentes de un servidor de aplicaciones deben ser accedidos exclusivamente por Servicios Distribuidos.
- **'El fin'** en mente, **no 'los medios'**.
- Identificar como **servicios-SOA** los componentes que tienen **valor empresarial** y son **reutilizables en diferentes aplicaciones** o los que necesariamente tienen que ser accedidos de forma remota (Porque la Capa de Presentación es remota, tipo cliente Windows).
- Si la capa de presentación es remota (p.e. WPF, Silverlight, OBA, etc.), hay que publicar mediante Servicios un 'Interfaz de Servicios Distribuidos'.
- Se consigue una meta de 'transparencia' e 'interoperabilidad'

Tabla 5.- Reglas globales de diseño

Regla Nº: D23	**La Arquitectura interna de un servicio debe de seguir las directrices de arquitectura N-Layer**

- **<u>Norma</u>**
- Cada servicio independiente, internamente debe de diseñarse según arquitectura en N-Capas (N-Layer), similar a la detallada en la presente guía.

Tabla 6.- Reglas globales de diseño

Regla Nº: D24	**Identificar la necesidad de uso de DTOs vs. Entidades del Dominio serializadas, como estructuras de datos a comunicar entre diferentes niveles físicos (Tiers)**

- **Norma**

- Esta regla significa que hay que identificar cuando merece la pena el sobre esfuerzo de implementar DTOs y Adaptadores DTOs versus hacer uso directo de entidades del Dominio serializadas.

- ☐En general, si la parte que consume nuestros servicios (cliente/consumidor) está bajo control del mismo equipo de desarrollo que los componentes de servidor, entoces será mucho mas productivo hacer uso de Entidades del Dominio serializadas. Sin embargo, si los consumidores son externos, desconocidos inicialmente y no están bajo nuestro control, el desacoplamiento ofrecido por los DTOs será crucial y realmente merecerá mucho la pena el trabajo sobre-esfuerzo de implementarlos.

Referencias:

Pros and Cons of Data Transfer Objects (Dino Esposito)
http://msdn.microsoft.com/en-us/magazine/ee236638.aspx

Building N-Tier Apps with EF4 (Danny Simons):
http://msdn.microsoft.com/en-us/magazine/ee335715.aspx

Tabla 7.- Reglas globales de diseño

Regla Nº: D25	Las fronteras de los Servicios deben ser explícitas

- **<u>Norma</u>**

- Quien desarrolla la **aplicación cliente** que consume un servicio debe de ser consciente de cuando y como consume remotamente un servicio para poder tener en cuenta escenarios de errores, excepciones, poco ancho de banda en la red, etc. Todo esto deberá de implementarse en los **Agentes de Servicio.**

- <u>Granularizar poco los interfaces</u> expuestos en los servicios (interfaces gruesos) permitiendo que se <u>minimice el número de llamadas a los servicios</u> desde las aplicaciones clientes.

- Mantener una **<u>máxima simplicidad en los interfaces de los servicios</u>**.

Tabla 8.- Reglas globales de diseño

Regla Nº: D26	Los Servicios deben ser Autónomos en Arquitecturas SOA puras

- **<u>Norma</u>**

- <u>En una Arquitectura SOA pura, los servicios deberían diseñarse, desarrollarse y versionarse, de forma autónoma.</u> Los servicios no deben depender fuertemente en su ciclo de vida con respecto a las aplicaciones que los consuman. Normalmente, esto requiere del uso de DTOs (Contratos de Datos).

- Los servicios deben de ofrecer ***ubicuidad***, es decir, ser ***localizables*** (mediante ***UDDI***) y sobre todo ser ***auto-descriptivos*** mediante estándares

como **WSDL y MEX** (***Metadata-Exchange***). Esto se consigue de forma sencilla desarrollando servicios con tecnologías que lo proporcionen directamente como ***ASMX y/o WCF***.

- La **seguridad**, especialmente la autorización, debe de ser **gestionada por cada servicio dentro de sus fronteras**. La autenticación, sin embargo, es recomendable que se pueda basar en sistemas de autenticación propagada, basado en estándares como ***WS-Federation***, etc.

Tabla 9.- Reglas globales de diseño

Regla Nº: D27

La Compatibilidad de los servicios se debe basar en Políticas

- **Norma**

- Implementar los requisitos horizontales y restricciones de compatibilidad a nivel de servicio (como **seguridad requerida, monitorización**, tipos de comunicaciones y protocolos, etc.) en forma de **políticas** (definidas en ficheros de configuración tipo **.config**) siempre que se pueda, en lugar de implementación de restricciones basadas en código (*hard-coded*).

Tabla 10.- Reglas globales de diseño

Regla Nº: D28

Contexto, Composición y Estado de Servicios SOA globales

- **Norma**

- Si los **Servicios-SOA** que estamos tratando son SERVICIOS GLOBALES (a ser consumidos por 'n' aplicaciones), entonces deben

diseñarse de forma que **ignoren el contexto desde el cual se les está 'consumiendo'**. No significa que los Servicios no puedan tener estados (*stateless*), significa que deberían ser independientes del contexto del consumidor, porque cada contexto consumidor será con toda probabilidad, diferente.

- '**Débilmente acoplados**' *(loosely coupled*): Los servicios-SOA que sean GLOBALES, pueden reutilizarse en 'contextos cliente' no conocidos en tiempo de diseño.

- **Se puede crear 'valor' al combinar Servicios** (p.e. reservar unas vacaciones, con un servicio reservar vuelo, con otro servicio reservar coche y con otro servicio reservar hotel).

Para mayor información sobre conceptos SOA generales y patrones a seguir, ver las siguientes referencias:

Referencias Servicios y SOA:

Service pattern
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpatterns/html/DesServiceInterface.asp

Service-Oriented Integration
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/archserviceorientedintegration.asp

19.- IMPLEMENTACIÓN DE LA CAPA DE SERVICIOS DISTRIBUIDOS CON WCF .NET 4.0

El objetivo del presente capítulo es mostrar las diferentes opciones que tenemos a nivel de tecnología para implementar la 'Capa de Servicios Distribuidos' y por supuesto, explicar la opción tecnológica elegida por defecto en nuestra Arquitectura Marco .NET 4.0, de referencia.

En el siguiente diagrama de Arquitectura resaltamos la situación de la Capa de Servicios Distribuidos:

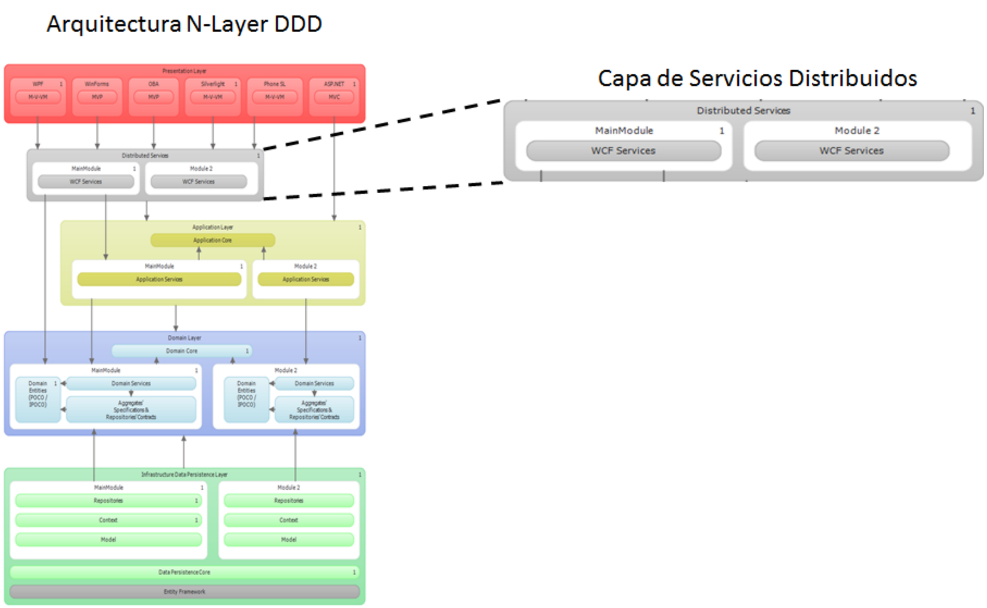


Figura 11.- Capa Servicios Distribuidos en Diagrama Layer con Visual Studio 2010

Para implementar Servicios Distribuidos con tecnología Microsoft, caben varias posibilidades, como analizaremos a continuación. Sin embargo, la tecnología más potente es WCF (*Windows Communication Foundation*), y es por lo tanto como recomendamos se implemente esta capa dentro de nuestra arquitectura propuesta.

20.- OPCIONES TECNOLÓGICAS

En plataforma Microsoft, actualmente podemos elegir entre dos tecnologías base orientadas a mensajes y Servicios Web:

- *ASP.NET Web Services (ASMX)*
- *Windows Communication Foundation (WCF)*

Así como otras tecnologías derivadas de más alto nivel:

- *Workflow-Services (''WCF+WF'')*
- *RAD (Rapid Application Development)*:
    - *WCF Data.Services (aka. ADO.NET DS)*
        - *Es la implementación de OData de Microsoft.*
    - *WCF RIA Services*

Sin embargo, para la presente arquitectura donde necesitamos desacoplamiento entre componentes de las diferentes capas, no nos es factible utilizar tecnologías de más alto nivel (RAD), por su fuerte acoplamiento extremo a extremo. Es por ello que las dos únicas opciones a plantearse inicialmente son las tecnologías base con las que podemos implementar Servicios-Web: WCF ó ASP.NET ASMX y en algunos casos *Workflow-Services*.

20.1.-Tecnología WCF

WCF proporciona una tecnología desacoplada en muchos sentidos (protocolos, formato de datos, proceso de alojamiento, etc.), permitiendo un control muy bueno de la configuración y contenido de los servicios. Considerar WCF en las siguientes situaciones:

- Los Servicios Web a crear requieren interoperabilidad con otras plataformas que también soportan SOAP y/o REST, como servidores de aplicación JEE
- Se requieren servicios Web no SOAP, es decir, basados en REST y formatos como RSS y ATOM *feeds*.

- Se requiere un máximo rendimiento en la comunicación y soporte tanto de mensajes SOAP como formato en binario para ambos extremos implementados con .NET y WCF

- Se requiere implementación de WS-Security para implementar autenticación, integridad de datos, privacidad de datos y cifrado en el mensaje.

- Se requiere implementación de WS-MetadataExchange en las peticiones SOAP para obtener información descriptiva sobre los servicios, como sus definiciones WSDL y políticas

- Se requiere implementación de 'WS-Reliable Messaging' para implementar comunicaciones confiables extremo a extremo, incluso realizándose un recorrido entre diferentes intermediarios de Servicios Web (No solo un origen y un destino punto a punto).

- Considerar WS-Coordination y WS-AT (Atomic Transaction) para coordinar transacciones *'two-phase commit'* en el contexto de conversaciones de Servicios Web.
  Ver: http://msdn.microsoft.com/en-us/library/aa751906.aspx

- WCF soporta varios protocolos de comunicación:

  - Para servicios públicos, Internet e interoperables, considerar HTTP

  - Para servicios con máx. Rendimiento y .NET extremo a extremo, considerar TCP

  - Para servicios consumidos dentro de la misma máquina, considerar named-pipes

  - Para servicios que deban asegurar la comunicación, considerar MSMQ, que asegura la comunicación mediante colas de mensajes.

20.2.-Tecnología ASMX (Servicios Web ASP.NET)

ASMX proporciona una tecnología más sencilla para el desarrollo de Servicios Web, si bien también es una tecnología más antigua y más acoplada/ligada a ciertas tecnologías, protocolos y formatos.

- Los Servicios Web ASP.NET se exponen mediante el servidor Web IIS

- Solo puede basarse en HTTP como protocolo de comunicaciones

- No soporta transacciones distribuidas entre diferentes servicios web

- No soporta los estándares avanzados de SOAP (WS-*), solo soporta SOAP WS-I Basic Profile
- Proporciona interoperabilidad con otras plataformas no .NET mediante SOAP WS-I, que es interoperable.

20.3.-Selección de tecnología

Para implementar servicios web simples, ASMX es muy sencillo de utilizar. Sin embargo, para el contexto que estamos tratando (Aplicaciones empresariales complejas orientadas al Dominio), **recomendamos encarecidamente el uso de WCF** por su gran diferenciamiento en cuanto a flexibilidad de opciones tecnológicas (estándares, protocolos, formatos, etc.) y en definitiva muchísimo más potente que servicios web .ASMX de ASP.NET.

20.4.- Tipos de Despliegue de Servicios WCF

La Capa de Servicios puede desplegarse en el mismo nivel físico (mismos servidores) que otras capas de la aplicación (Capa del Dominio e incluso capa de presentación Web basada en ASP.NET) o bien en niveles separados (otros servidores) si lo demandan razones de seguridad y/o escalabilidad, que deben estudiarse caso a caso.

En la mayoría de los casos, la Capa de Servicios residirá en el mismo nivel (Servidores) que las Capas de Dominio, Aplicación, Infraestructura, etc. para maximizar el rendimiento, puesto que si separamos las capas en diferentes niveles físicos estamos añadiendo latencias ocasionadas por las llamadas remotas. Considerar las siguientes guías:

- **Desplegar la Capa de Servicios en el mismo nivel físico que las Capas de Dominio, Aplicación, Infraestructura**, etc., para mejorar el rendimiento de la aplicación, a menos que existan requerimientos de seguridad que lo impida. Este es el caso más habitual para arquitecturas N-Tier con clientes RIA y Rich.

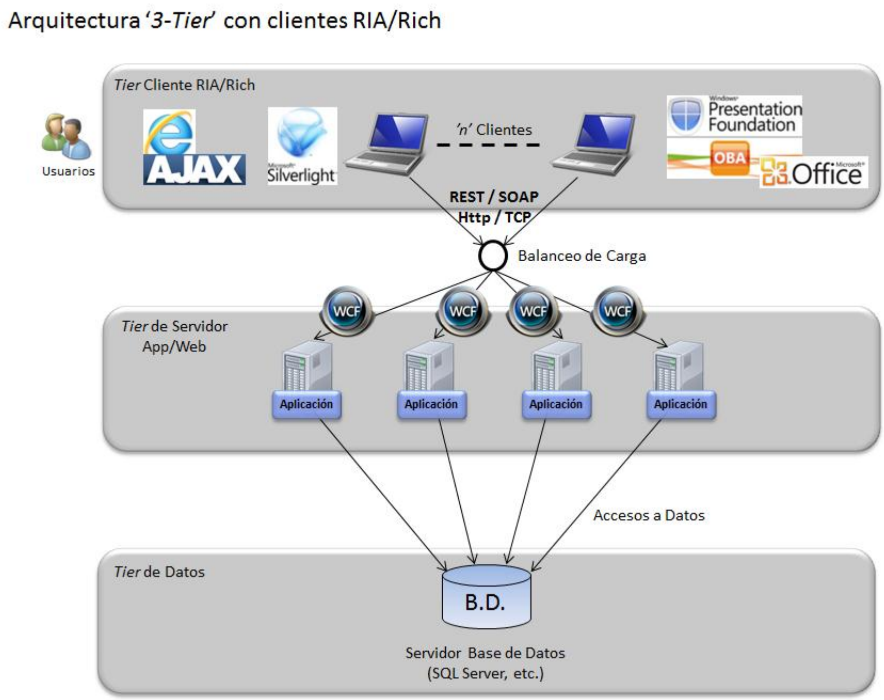


Figura 12.- Clientes RIA/Rich accediendo remotamente a Servicios WCF

- **Desplegar la Capa de Servicios en el mismo nivel físico que la Capa de Presentación, si ésta es una Capa de presentación Web tipo ASP.NET**, para mejorar el rendimiento de la aplicación. El separarlo de la capa Web ASP.NET solo debe hacerse por razones de escalabilidad no muy comunes y que deben demostrarse. Si los Servicios Web se localizan en el mismo nivel físico que el consumidor, considerar el uso de *named-pipes*. Si bien, otra opción en ese caso es no hacer uso de Servicios Web y utilizar directamente los objetos a través del CLR. Esta es probablemente la opción con la que conseguimos mejor rendimiento. No tiene demasiado sentido hacer uso de Servicios Distribuidos si los consumimos desde dentro de la misma máquina. Como dice **Martin Fowler**: '*La primera ley de la programación distribuida es "No distribuyas" (a no ser que realmente lo necesites)*'. Pero es posible que, por homogeneidad, no querer mantener varias versiones del mismo software, y querer mantener un software completamente SOA, se quiera realizar este enfoque.

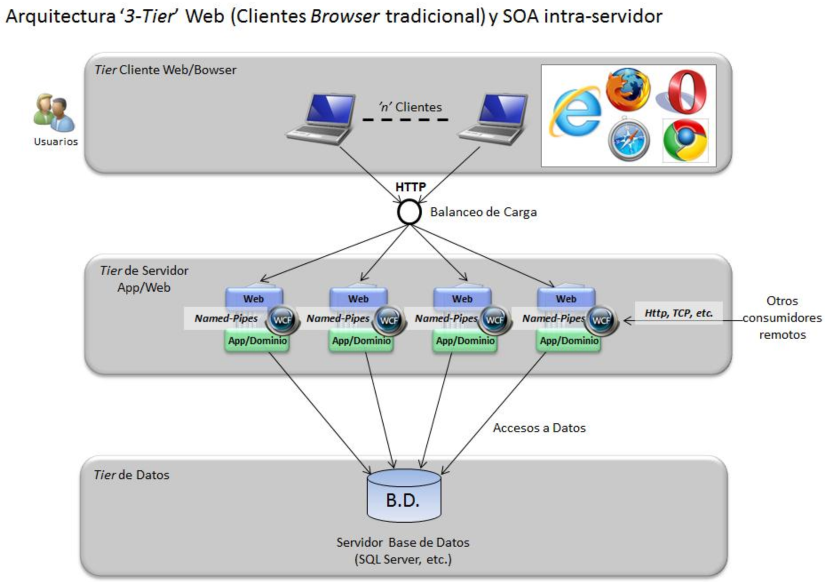


Figura 13.- App-Web con Servicios WCF intra-servidor reutilizables para llamadas externas desde otros consumidores remotos

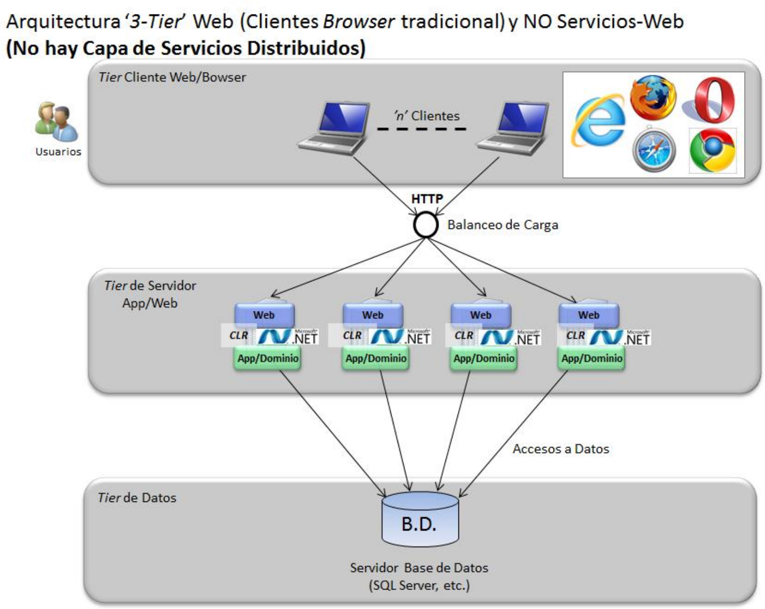
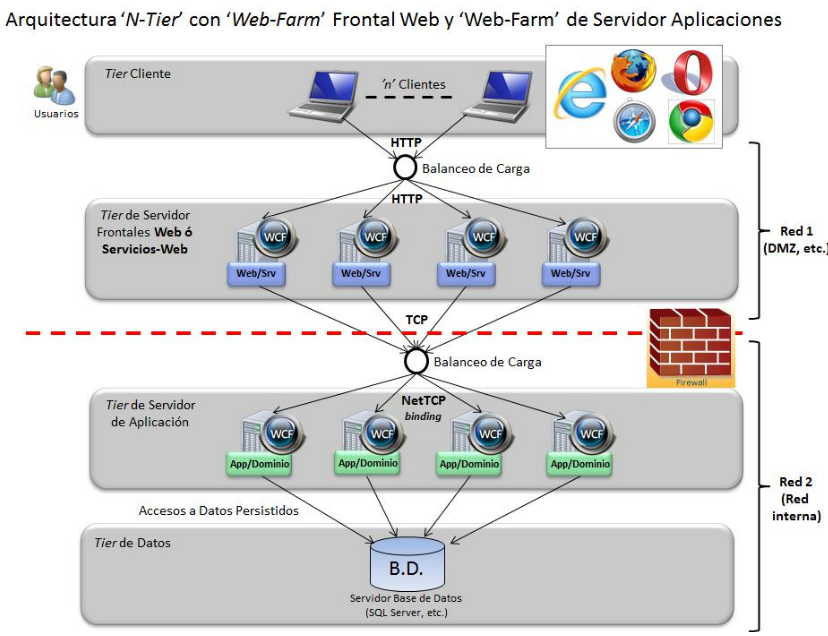


Figura 14.- App-Web sin Capa de Servicios Distribuidos

- **Desplegar la Capa de Servicios en diferente nivel físico que la Capa de Presentación Web.** Hay situaciones en las que separar los frontales web visuales de los frontales de servicios web puede aumentar la escalabilidad, si bien, debe demostrarse con pruebas de carga. Cualquier introducción de comunicaciones remotas es, por defecto, un motivo de perdida de rendimiento debido a la latencia introducida en las comunicaciones, por lo que debe probarse lo contrario si se desea separar capas en diferentes servidores físicos. Otra razón por la que separar el frontal Web visual (ASP.NET) del Servidor de aplicaciones (Servicios Web) pueden ser razones de seguridad, diferentes redes públicas y privadas para componentes internos. En ese caso, no hay problema en separar estos niveles.



- Si los consumidores son aplicaciones .NET, dentro de la misma red interna y se busca el máximo rendimiento, considerar el *binding* TCP en WCF para las comunicaciones.

- Si el servicio es público y se requiere interoperabilidad, hacer uso de HTTP

21.- INTRODUCCIÓN A WCF (WINDOWS COMMUNICATION FOUNDATION)

'Windows Communication Foundation' (denominado '***Indigo***' anteriormente en su fase BETA en los años 2004-2005) es la plataforma estratégica de tecnologías .NET para desarrollar '**sistemas conectados**' (Aplicaciones distribuidas, etc.). Es una plataforma de infraestructura de comunicaciones construida a partir de la evolución de arquitecturas de Servicios-Web. El soporte de Servicios avanzados en WCF proporciona una mensajería programática que es **segura**, **confiable**, **transaccional** e **interoperable** con otras plataformas (Java, etc.). En su mayoría WCF está diseñado siguiendo las directrices del modelo ***'Orientado a Servicios' (SOA)***. Por último, WCF unifica todas las diferentes tecnologías de sistemas distribuidos que disponía Microsoft en una única **arquitectura componentizable**, **desacoplada** y **extensible**, pudiendo **cambiar** de forma declarativa **protocolos de transporte**, **seguridad**, patrones de **mensajería**, tipo de **serialización** y modelos de **'hosting'** (albergue).

Es importante resaltar que WCF rediseñada desde cero, no está basado en ASMX 2.0 (Servicios Web básicos de ASP.NET). Es realmente mucho más avanzado que ASMX 2.0.

WCF forma parte a su vez de ***.NET Framework, desde la versión 3.0 de .NET (año 2006)***.

A continuación mostramos un esquema de la evolución y unificación de los *stacks* de protocolos de Microsoft, como he comentado anteriormente:



Figura 15.- Evolución y unificación de los stacks

Los objetivos prácticos de WCF es que no se tengan que tomar decisiones de diseño y arquitectura sobre tecnologías distribuidas (**ASMX** vs. **Remoting** vs. **WSE**, etc.), dependiendo del tipo de aplicación. Esto es algo que teníamos que hacer antes de la

aparición de WCF, pero a veces, los requerimientos de la aplicación cambiaban y podían aparecer problemas de aspectos no soportados por la tecnología elegida inicialmente.

El **objetivo** principal de WCF es **poder realizar una implementación de cualquier combinación de requerimientos con una <u>única plataforma tecnológica de comunicaciones</u>**.

En la siguiente tabla se muestran las diferentes características de las diferentes tecnologías anteriores y como con WCF se unifican en una sola tecnología:

Tabla 11.- Características tecnologías de comunicaciones

	ASMX	.NET Remoting	Enterprise Services	WSE	MSMQ	WCF
Servicios Web Básicos Interoperables	X					X
Comunicación .NET – .NET		X				X
Transacciones Distribuidas, etc.			X			X
Especificaciones WS-*				X		X
Colas de Mensajes					X	X

Por ejemplo, si se quiere desarrollar un *Servicio-Web* con comunicaciones confiables, que soporte sesiones y propagación de transacciones entre diferentes servicios e incluso extenderlo dependiendo de los tipos de mensajes que llegan al sistema, esto puede hacerse con WCF. Hacer esto sin embargo con tecnologías anteriores no es del todo imposible, pero requeriría mucho tiempo de desarrollo y un alto conocimiento de todas las diferentes tecnologías de comunicaciones (y sus diferentes esquemas de programación, etc.). Por lo tanto, otro objetivo de WCF es ser más productivo no solamente en los desarrollos iniciales sino también en los desarrollos que posteriormente tienen cambios de requerimientos (funcionales y técnicos), puesto que solamente se tendrá que conocer un único modelo de programación que unifica todo lo positivo de ASMX, WSE, Enterprise Services (COM+), MSMW y .Net Remoting. Además, como aspectos nuevos, no hay que olvidar que WCF es la 'implementación estrella' de Microsoft de las especificaciones WS-*, las cuales han sido elaboradas y estandarizadas por diferentes fabricantes (incluyendo a Microsoft) durante los últimos cinco o seis años, con el objetivo de conseguir una interoperabilidad real a lo largo de diferentes plataformas y lenguajes de programación (.NET, Java, etc.) pudiendo, sin embargo, realizar aspectos avanzados (cifrado, firma, propagación de transacciones entre diferentes servicios, etc.).

Por último, cabe destacar que **WCF es interoperable,** bien basándonos en los estándares SOAP de ***WS-****, o bien basándonos en REST.

21.1.-El 'ABC' de Windows Communication Foundation

Las siglas 'ABC' son claves para WCF porque coinciden con aspectos básicos de cómo están compuestos los '*End-Points*' de un Servicio WCF.

Los '*End-Points*' son básicamente los extremos en las comunicaciones basadas en WCF y por lo tanto también los puntos de entrada a los servicios. Un '*End-Point*' es internamente bastante complejo pues ofrece diferentes posibilidades de comunicación, direccionamiento, etc.

Precisamente y volviendo a las siglas 'ABC' como algo para recordar, un '*End-Point*' está compuesto por 'ABC', es decir:

- "**A**" para ***'Address'*** (Dirección): ¿**Dónde** está el servicio situado?
- "**B**" para ***'Binding'*** (Enlace): ¿**Cómo** hablo con el servicio?
- "**C**" para ***'Contract'*** (Contrato): ¿**Qué** me ofrece el servicio?

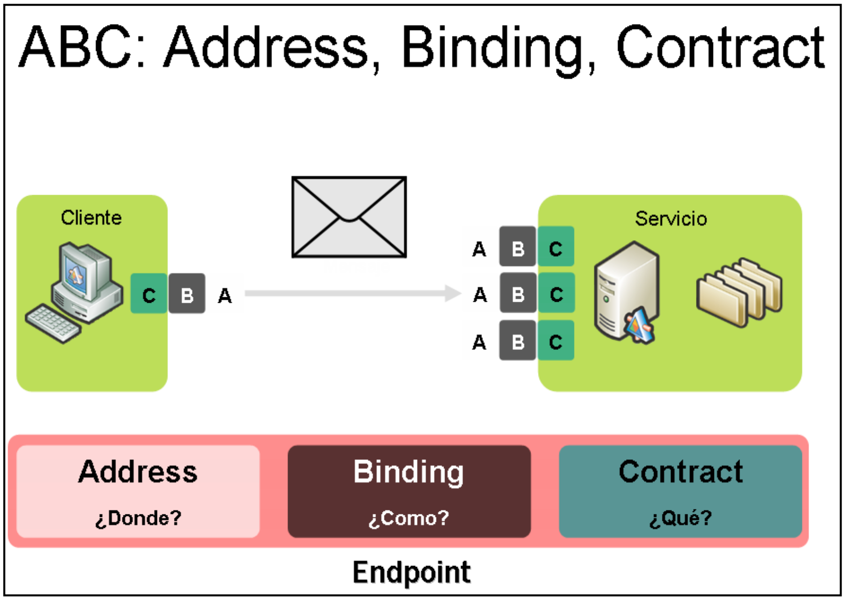


Figura 16.- Address, Binding, Contract

Es importante destacar que estos tres elementos son independientes y existe un gran desacoplamiento entre ellos. Un contrato puede soportar varios *'bindings'* y un *'binding'* puede soportar varios contratos. Un Servicio puede tener muchos '*endpoints'* (contrato enlazado a dirección) coexistiendo y disponibles al mismo tiempo. Por ejemplo, se puede exponer un servicio via HTTP y SOAP 1.1 para ofrecer máxima interoperabilidad y al mismo tiempo, exponerlo via TCP y formato binario para ofrecer

un rendimiento máximo, teniendo como resultado dos 'end-points' que pueden residir simultáneamente sobre el mimo Servicio.

Dirección (*Address*)

Al igual que una página web o un webservice, todos los servicios WCF deben tener una dirección. Lo que sucede, es que a diferencia de los anteriores, un servicio WCF puede proporcionar direcciones para los siguientes protocolos:

- HTTP
- TCP
- *NamedPipes*
- PeerToPeer (P2P)
- MSMQ

Enlace (*Binding*)

Un **binding** especifica cómo se va a acceder al servicio. Define, entre otros, los siguientes conceptos:

- Protocolo de transporte empleado: HTTP, TCP, NamedPipes, P2P, MSMQ, etc.
- Codificación de los mensajes: texto plano, binario, etc.
- Protocolos WS-* a aplicar: soporte transaccional, seguridad de los mensajes, etc.

Contrato (*Contract*)

El contrato del servicio representa el interfaz que ofrece ese servicio al mundo exterior. Por tanto, ahí se definen los métodos, tipos y operaciones que se desean exponer a los consumidores del servicio. Habitualmente el contrato de servicio se define como una clase de tipo interfaz a la que se le aplica el atributo *ServiceContractAttribute*. La lógica de negocio del servicio se codifica implementando el interfaz antes diseñado.

En el siguiente esquema se muestra la arquitectura simplificada de componentes de WCF:

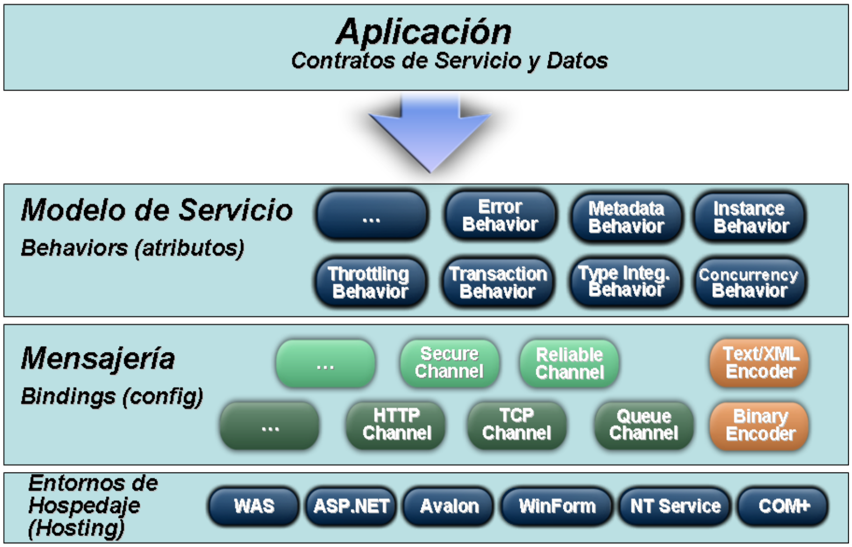


Figura 17.- Arquitectura simplificada componentes WCF

Y en este otro diagrama se puede observar el **desacoplamiento** y **combinaciones** de ***WCF***:

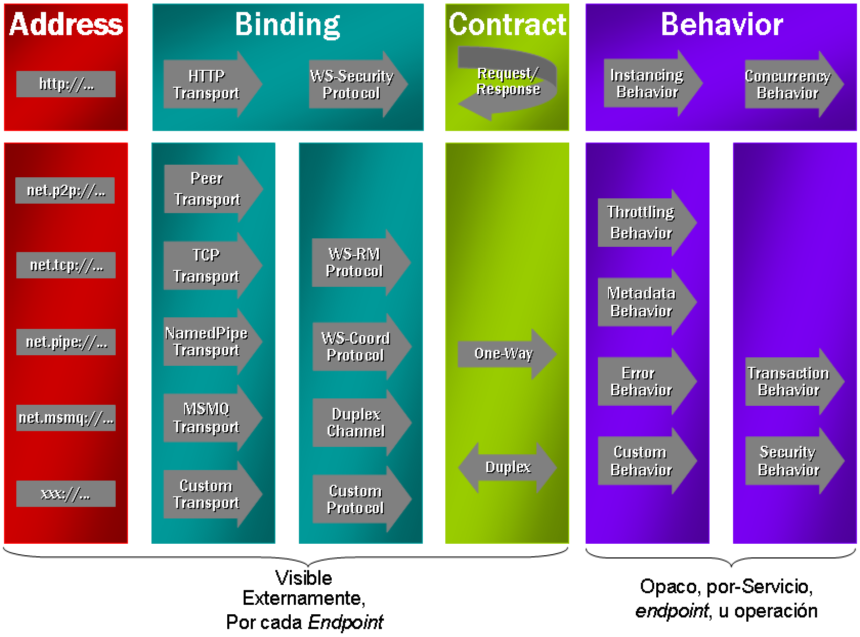


Figura 18.- Desacoplamiento y Combinaciones de WCF

"ABC" significa también que el desarrollo y configuración de un servicio WCF se hace en tres pasos:

1.- Se define el contrato y su implementación.

2.- Se configura un binding que selecciona un transporte junto con características de calidad de servicio, seguridad y otras opciones. Equivale a la 'Mensajería' y 'Bindings' del diagrama anterior).

3.- Se despliega el servicio y el 'endpoint' en un entorno de hospedaje (Proceso de ejecución. Equivale a los 'Entornos de Hosting' del diagrama anterior).

4.- Vamos a ver estos pasos en detalle en los siguientes apartados.

21.2.-Definición e implementación de un servicio WCF

El desarrollo de un servicio WCF básico, es decir, que simplemente implementamos comunicación entre el cliente y el servicio, es relativamente sencillo. No llega a ser tan sencillo como el desarrollar un servicio web básico en ASP.NET (ASMX), porque el desacoplamiento que tiene WCF (y sobre el que siempre hago tanto hincapié) tiene cierto coste de implementación, pero veréis que no es mucho más complicado. Por otro lado, dicho desacoplamiento por beneficia muchísimo para el futuro, pudiendo cambiar características clave de nuestro desarrollo (protocolo de comunicaciones, formato, etc.) cambiando solamente declaraciones y sin necesidad de cambiar nuestro modelo de programación ni tecnología.

Vamos a ir viendo estos pasos.

Definición del Contrato de un Servicio WCF

Tanto el contrato como la implementación de un servicio de realiza en una librería de clases .NET (.DLL). Equivale al '***Modelo de Servicio***' del diagrama anteriormente mostrado.

Los contratos de Servicio se modelan en .NET utilizando definiciones de interfaces tradicionales de C# (o VB.NET). Puedes utilizar cualquier interfaz .NET como punto de arranque, como el mostrado a continuación:

```
namespace MiEmpresa.MiAplicacion.MiServicioWcf
{
    public interface ISaludo
    {
        string Saludar(string nombre);
    }
}
```

Para convertir este interfaz normal .NET en un contrato de servicio, tenemos que simplemente 'decorarlo' con ciertos atributos, en concreto al propio interfaz con el atributo [ServiceContract] y a cada método que quieras exponer como una operación del servicio, con el atributo [OperationContract], como se muestra a continuación:

```
using System.ServiceModel;

namespace MiEmpresa.MiAplicacion.MiServicioWcf
{
    [ServiceContract]
    public interface ISaludo
    {
        [OperationContract]
        string Saludar(string nombre);
    }
}
```

Estos atributos influyen en el mapeo entre los mundos .NET y SOAP. WCF utiliza la información que encuentra en el contrato del servicio para realizar el envío y la serialización. El envío ('*dispatching*') es el proceso de decidir qué método llamar para cada mensaje SOAP de entrada. La Serialización es el proceso de mapeo entre los datos encontrados en el mensaje SOAP y los objetos .NET correspondientes utilizados en la invocación del método. Este mapeo lo controla un contrato de datos de operación. WCF envía mensajes basándose en el '*action*' del mensaje. Cada método en un contrato de servicio se le asigna automáticamente un valor de acción ('*action*') basándose en el *namespace* del servicio y en el nombre del método.

Se puede llegar a implementar un servicio WCF con una única clase (sin interfaz), pero es bastante recomendable separar el contrato en un interfaz y situar en la clase simplemente la implementación interna del servicio. Esto ofrece varias ventajas como:

- Permite modificar la implementación del servicio sin romper el contrato.
- Facilita el versionado de los servicios estableciendo nuevos interfaces.
- Un interfaz puede extender/heredar de otro interfaz.
- Una única clase puede implementar varios interfaces.

Implementación del Servicio WCF

Ahora podemos desarrollar el servicio (el código que queremos que se ejecute), simplemente implementando el interfaz .NET en una clase .NET:

```
using System.ServiceModel;

namespace MiEmpresa.MiAplicacion.MiServicioWcf
{
    public class Saludo : ISaludo
    {
        public string ISaludo.Saludar(string nombre)
        {
```

```
                return "Bienvenido a este libro " + nombre;
            }
        }
    }
```

Haciendo esto, se garantiza que la clase **Saludo** soporta el contrato de servicio definido por el interfaz **ISaludo**. Cuando se utiliza un interfaz para definir un contrato de servicio, no necesitamos aplicar sobre la clase ningún atributo relacionado con el contrato. Sin embargo, podemos utilizar atributos de tipo **[ServiceBehavior]** para influenciar su comportamiento/ejecución local:

```
using System.ServiceModel;

namespace MiEmpresa.MiAplicacion.MiServicioWcf
{
    ... // Omito la definición del interfaz

    [ServiceBehavior(
       InstanceContextMode=InstanceContextMode.Single,
       ConcurrencyMode=ConcurrencyMode.Multiple)]
    public class Saludo : ISaludo
    {
       ...
```

Este ejemplo en particular le dice a WCF que gestione una instancia *singleton* (se comparte el mismo objeto instanciado del servicio entre todos los clientes) y además que se permite acceso *multi-thread* a la instancia (deberemos por lo tanto controlar los accesos concurrentes a las zonas de memoria compartida de dicho objeto, mediante secciones críticas, semáforos, etc.).
También existe un atributo **[OperationBehavior]** para controlar comportamientos a nivel de de operación/método.

Los comportamientos ('*behaviors*') influyen en el procesamiento dentro del *host* (proceso de ejecución del servicio, posteriormente lo vemos en detalle), pero no tienen impacto de ningún tipo en el contrato del servicio. Los comportamientos son uno de los principales puntos de extensibilidad de WCF. Cualquier clase que implemente **IServiceBehavior** puede aplicarse a un servicio mediante el uso de un atributo propio ('*custom*') o por un elemento de configuración.

Definición de contratos de datos (Data Contracts)

A la hora de llamar a un servicio, WCF serializa automáticamente los parámetros de entrada y salida estándar de .NET (tipos de datos básicos como string, int, double, e incluso tipos de datos más complejos como DataTable y DataSet.)

Sin embargo, en muchas ocasiones, nuestros métodos de WCF tienen como parámetros de entrada o como valor de retorno clases definidas en nuestro código (*clases entidad custom* o propias nuestras). Para poder emplear estas clases entidad *custom* en las operaciones/métodos de WCF, es requisito imprescindible que sean serializables. Para ello, el mecanismo recomendado consiste en marcar la clase con el

atributo **DataContract** y las propiedades de la misma con el atributo **DataMember**. Si se desea no hacer visible una propiedad, bastará con no marcar la misma con el atributo **DataMember.**

A modo ilustrativo se modifica el ejemplo anterior modificando el método *Saludor()* para que tome como parámetro de entrada la clase *PersonaEntidad*.

```
using System.ServiceModel;
namespace MiEmpresa.MiAplicacion.MiServicioWcf
{
    //CONTRATO DEL SERVICIO
    [ServiceContract]
    public interface ISaludo
    {
        [OperationContract]
        string Saludar(PersonaEntidad persona);
    }

    //SERVICIO
    public class Saludo : ISaludo
    {
        public string ISaludo.Saludar(PersonaEntidad persona)
        {
            return "Bienvenido a este libro " + persona.Nombre + " " +
persona.Apellidos;
        }
    }
    // CONTRATO DE DATOS
    [DataContract]
    public class PersonaEntidad
    {
        string _nombre;
        string _apellidos;

        [DataMember]
        public string Nombre
        {
            get { return _nombre; }
            set { _nombre = value; }
        }
        [DataMember]
        public string Apellidos
        {
            get { return _apellidos; }
            set { _apellidos = value; }
        }
    }

}
```

21.3.-Hospedaje del servicio (Hosting) y configuración (Bindings)

Una vez hemos definido nuestro servicio, debemos seleccionar un host (proceso) donde se pueda ejecutar nuestro servicio (hay que recordar que hasta ahora nuestro servicio es simplemente una librería de clases .NET, una .DLL que por sí sola no se puede ejecutar). Este proceso 'hosting' puede ser casi cualquier tipo de proceso, puede ser IIS, una aplicación de consola, un Servicio Windows, etc.

Para desacoplar el host del propio servicio, es interesante crear un nuevo proyecto en la solución de Visual Studio, que defina nuestro host. Según el tipo de *hosting*, se crearán distintos proyectos:

- Hosting en IIS/WAS: proyecto **Web Application**
- Hosting en ejecutable app-consola: proyecto tipo **Console Application**
- Hosting en 'Servicio Windows': proyecto tipo **Windows Service**

La flexibilidad de WCF permite tener varios proyectos de host albergando un mismo servicio. Esto es útil ya que durante la etapa de desarrollo del servicio podemos albergarlo en una aplicación de tipo *consola* lo que facilita su depuración, pero posteriormente, se puede añadir un segundo proyecto de tipo web para preparar el despliegue del servicio en un IIS o en un servicio Windows.

A modo de ejemplo, vamos a crear un proyecto de consola como proceso *hosting* y le añadiremos una referencia a la librería **System.ServiceModel**.

A continuación, en la clase Main se deberá instanciar el objeto **ServiceHost** pasándole como parámetro el tipo de nuestro servicio *Saludo.*

```
using System.ServiceModel;
static void Main(string[] args)
{
    Type tipoServicio = typeof(Saludo);
    using (ServiceHost host = new ServiceHost(tipoServicio))
    {
        host.Open();

        Console.WriteLine("Está    disponible    el    Servicio-WCF    de
Aplicación.");
        Console.WriteLine("Pulse una tecla para cerrar el servicio");
        Console.ReadKey();

        host.Close();
    }
}
```

En el ejemplo anterior se observa cómo se define el host del servicio, posteriormente se inicia el mismo y se queda en ejecución 'dando servicio' hasta que el usuario pulse una tecla, en cuyo momento finalizaríamos el servicio:

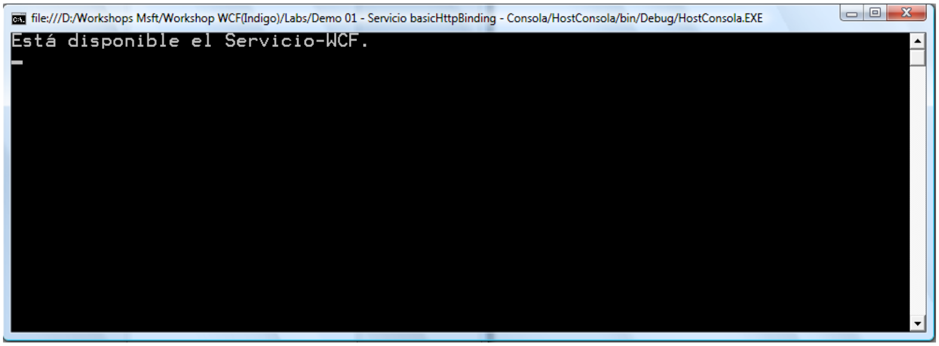


Figura 19.- Servicio WCF

Esta ejecución de aplicación de consola haciendo hosting del servicio WCF realmente no podemos realizarlo hasta que tengamos también el servicio configurado, mediante las secciones XML del .config, que vamos a ver en unos momentos. Una vez que hayamos configurado el host con la información de *endpoints* (como veremos), WCF puede entonces construir el *runtime* necesario para soportar nuestra configuración. Esto ocurre en el momento en que se llama a Open() en un ServiceHost particular, como podemos ver en el código C# de arriba (host.Open();)

Una vez que finaliza el método Open(), el runtime de WCF está ya construido y preparado para recibir mensajes en la dirección especificada por cada endpoint. Durante este proceso, WCF genera un escuchador de *endpoint* por cada *endpoint* configurado. (Un escuchador *endpoint* o *endpoint-listener* es la pieza de código que realmente escucha y espera mensajes de entrada utilizando el transporte especificado y la dirección).

Se puede obtener información sobre el servicio en tiempo de ejecución mediante el modelo de objetos expuesto por la clase **ServiceHost**. Esta clase permite obtener cualquier cosa que se quisiera conocer sobre el servicio inicializado, como qué *endpoints* expone y qué *endpoints listeners* están actualmente activos.

Quiero resaltar que el uso de 'aplicaciones de consola' como proceso *hosting* de servicios-WCF debe de utilizarse solamente para pruebas de concepto, demos, y servicios de pruebas, pero nunca, lógicamente, para servicios-WCF en producción. Un *deployment* de un servicio-WCF en un entorno de producción normalmente debería realizarse en un proceso *hosting* de alguno de los siguientes tipos:

Tabla 12.- Posibilidades de Alojamiento de Servicios WCF

Contexto	Proceso Hosting	Protocolos	Sistema Operativo req.
Cliente-Servicio	IIS 6.0	http/https	- Windows Server 2003 - Windows XP - (o versión superior Windows)
Cliente-Servicio	Servicio Windows	Tcp named-pipes msmq http/https	- Windows Server 2003 - Windows XP - (o versión superior Windows)
Cliente-Servicio	WAS-IIS7.x AppFabric	Tcp named-pipes msmq http/https	- Windows Server 2008 o superior
			- Windows Vista
Peer-To-Peer	WinForms ó WFP client	Peer-to-Peer	- Windows Server 2003 - Windows XP - (o versión superior Windows)

21.4.-Configuración de un servicio WCF

Sin embargo, para que el servicio sea funcional, es necesario configurarlo antes. Esta configuración se puede hacer de dos formas:

- Configuración del servicio en el propio código (*hard-coded*).

- Configuración del servicio mediante ficheros de configuración (**.config*). Esta es la opción más recomendable, puesto que ofrece flexibilidad para cambiar parámetros del servicio como su dirección, protocolo, etc. sin necesidad de recompilar, también por lo tanto facilita el despliegue del servicio y por último, aporta una mayor simplicidad ya que permite emplear la utilidad **Service Configuration Editor** incluida en el SDK de Windows.

Dentro del modelo '**ABC**', hasta ahora ya habíamos definido el contrato (interfaz) la implementación (clase), incluso el proceso *hosting*, pero ahora es necesario asociar ese contrato con un *binding* concreto y con una dirección y protocolo. Esto lo hacemos normalmente dentro del fichero .config (app.config si es un proceso nuestro, o web.config si es IIS quien hace *hosting*).

Así pues, un ejemplo muy sencillo del XML de un fichero app.config, sería el siguiente:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <services>
            <service
name="MiEmpresa.MiAplicacion.MiServicioWcf.Saludo">
                <endpoint

address="http://localhost:8000/ServicioSaludo/http/"
                    binding="basicHttpBinding"

contract="MiEmpresa.MiAplicacion.MiServicioWcf.ISaludo"
                    bindingConfiguration="" />
            </service>
        </services>
    </system.serviceModel>
</configuration>
```

Por supuesto, el app.config ó web.config puede tener otras secciones adicionales de XML. En el app.config de arriba, podemos ver claramente el '**ABC de WCF**': **Address, Binding y Contract.**

Es importante resaltar que en este ejemplo nuestro servicio está 'escuchando' y ofreciendo servicio por HTTP (en concreto por http://localhost:8000) y sin embargo no estamos utilizando un servidor web como IIS, es simplemente una aplicación de consola o podría ser también un servicio Windows que ofreciera también servicio por HTTP. Esto es así porque WCF proporciona integración interna con **HTTPSYS**, que permite a cualquier aplicación convertirse automáticamente en un ***http-listener***.

La configuración del fichero .config podemos hacerla manualmente, escribiendo directamente dicho XML en el .config (recomiendo hacerlo así, porque es cuando realmente se va aprendiendo a utilizar los *bindings* y sus configuraciones) o bien, si estamos comenzando con WCF o incluso si hay cosas que no nos acordamos exactamente como sería la configuración del XML, podemos hacerlo mediante un asistente de Visual Studio. En Visual Studio 2008 tenemos directamente este asistente.



Figura 20.- Asistente configuración WCF de Visual Studio

Para ello, en nuestro proyecto de consola del ejemplo, y una vez hayamos añadido un fichero *app.config*, a continuación, pulsando con el botón derecho sobre dicho fichero .config, se selecciona *"Edit WCF Configuration…"* y se mostrará la ventana del **Service Configuration Editor**.

22.- IMPLEMENTACIÓN DE CAPA DE SERVICIOS WCF EN ARQUITECTURA N-LAYER

En nuestra solución ejemplo, tendríamos un árbol similar al siguiente, donde resaltamos la situación de los proyectos que implementan el Servicio WCF:

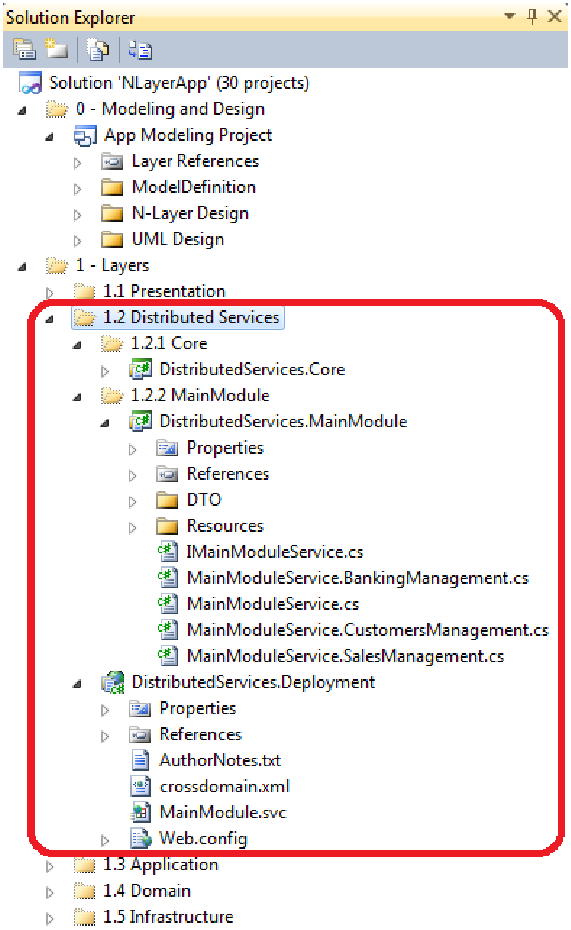


Figura 21.- Árbol situación proyectos servicio WCF

La Capa de Servicios Distribuidos estará compuesta para cada aplicación, como se puede apreciar, por un único proyecto de *hosting* que llamamos '*DistributedServices.Deployment*' (por defecto uno, pero dependiendo de necesidades, podríamos llegar a tener varios tipos de *hosting*) que en el ejemplo hemos elegido un proyecto Web que se alojará en un servidor de IIS (O Cassini en entorno de desarrollo).

Por cada módulo vertical de la aplicación, dispondremos de un *assembly* de implementación de Servicio WCF. En este caso, disponemos de un solo módulo vertical de aplicación, por lo que tenemos un solo *assembly* de implementación de Servicio WCF, llamado en este ejemplo como "*DistributedServices.MainModule*".

Adicionalmente podemos tener librerías de clases donde dispongamos de código reutilizable para los diferentes servicios de diferentes módulos respectivos. En este ejemplo, disponemos de una librería denominada "*DistributedServices.Core*", donde básicamente tenemos cierto código de Gestión de Errores y *Faults* de WCF.

23.- TIPOS DE OBJETOS DE DATOS A COMUNICAR CON SERVICIOS WCF

Como explicamos en el capítulo de Servicios Distribuidos (a nivel lógico), intentando unificar opciones, los tipos de objetos de datos a comunicar, más comunes, a pasar de un nivel a otro nivel remoto dentro de una Arquitectura N-Tier son:

- Valores escalares
- DTOs (*Data Transfer Objects*)
- Entidades del Dominio serializadas
- Conjuntos de registros (Artefactos desconectados)

Estas opciones, a nivel de tecnología veríamos el siguiente mapeo:

Tabla 13.- Opciones Mapeo

Tipo de Objeto Lógico	Tecnología .NET
Valores escalares	- String, int, etc.
DTOs (Data Transfer Objects)	- Clases .NET propias con estructuras de datos
Entidades del Dominio serializadas dependientes de la tecnología de infraestructura de acceso a datos	- Entidades simples/nativas de Entity Framework (Era la única posibilidad directa en EF 1.0)
Entidades del Dominio serializadas, NO dependientes de la tecnología de infraestructura de acceso a datos	- **Entidades POCO de Entity Framework** (Disponibles a partir de EF 4.0) - **Entidades *SELF-TRACKING* IPOCO de EF** (Disponibles a partir de EF 4.0)
Conjuntos de registros dependientes de la tecnología de infraestructura de acceso a datos. (Artefactos desconectados)Concepto 1	- **DataSets, DataTables** de ADO.NET

A pesar de que las Entidades simples/nativas normalmente no es el mejor patrón para las aplicaciones de N-Tier (tienen una dependencia directa de la tecnología, de EF), era la opción más viable en la primera versión de EF. Sin embargo, EF4 cambia significativamente las opciones para la programación de N-Tier. Algunas de las nuevas características son:

1.- Nuevos métodos que admiten operaciones desconectadas, como *ChangeObjectState* y *ChangeRelationshipState*, que cambian una entidad o una relación a un estado nuevo (por ejemplo, añadido o modificado); *ApplyOriginalValues*, que permite establecer los valores originales de una entidad y el nuevo evento *ObjectMaterialized* que se activa cada vez que el *framework* crea una entidad.

2.- Compatibilidad con Entidades POCO y valores de claves extranjeras en las entidades. Estas características nos permiten crear clases entidad que se pueden compartir entre la implementación de los componentes del Servidor (Modelo de Dominio) y otros niveles remotos, que incluso puede que no tengan la misma versión de *Entity Framework* (.NET 2.0 o Silverlight, por ejemplo). Las Entidades POCO con claves extranjeras tienen un formato de serialización sencillo que simplifica la interoperabilidad con otras plataformas como JAVA. El uso de claves externas también permite un modelo de concurrencia mucho más simple para las relaciones.

3.- Plantillas T4 para personalizar la generación de código de dichas clases POCO ó STE (*Self-Tracking Entities*).
El equipo de Entity Framework ha usado estas características también para implementar el patrón de Entidades STE (Self-Tracking Entities) en una plantilla T4, con lo que ese patrón es mucho más fácil de usar pues tendremos código generado sin necesidad de implementarlo desde cero.

Con estas nuevas capacidades en EF 4.0, normalmente a la hora de tomar decisiones de diseño y tecnología sobre tipos de datos que van a manejar los Servicios Web, deberemos situar en 'una balanza' aspectos de **Arquitecturas puristas** (desacoplamiento entre entidades del Dominio de datos manejados en Capa de Presentación, Separación de Responsabilidades, Contratos de Datos de Servicios diferentes a Entidades del Dominio) **versus Productividad y *Time-To-Market*** (gestión de concurrencia optimista ya implementada por nosotros, no tener que desarrollar conversiones entre DTOs y Entidades, etc.).

Si situamos los diferentes tipos de patrones para implementar objetos de datos a comunicar en aplicaciones N-Tier (datos a viajar por la red gracias a los Servicios Web), tendríamos algo así:

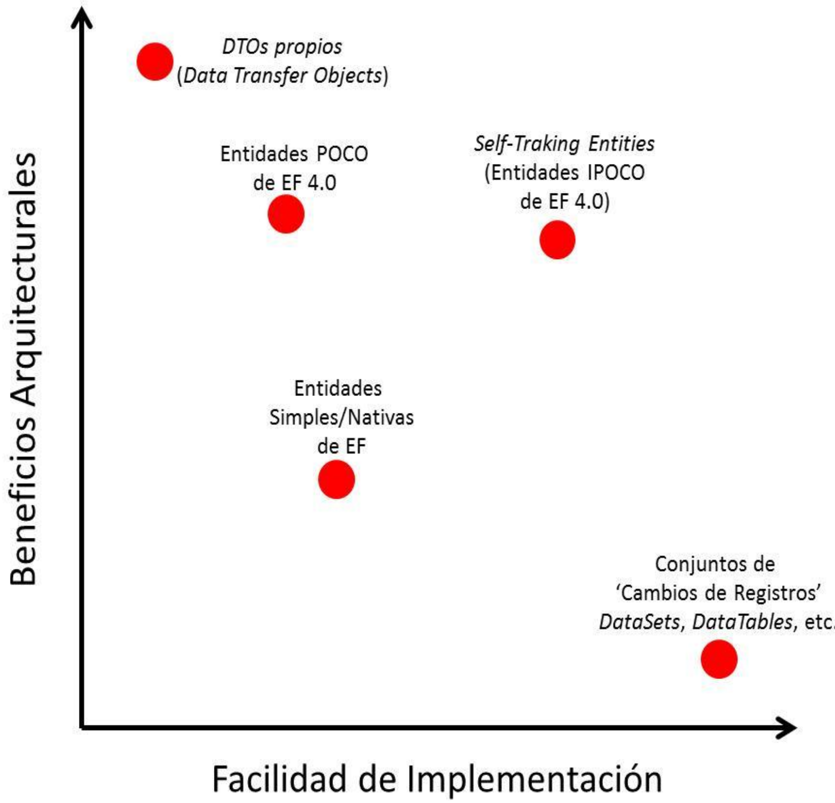


Figura 22.- Balance entre Facilidad de Implementación y Beneficios Arquitecturales

El patrón correcto para cada situación en particular, depende de muchos factores. En general, los DTOs (como introdujimos a nivel lógico en el capítulo de 'Capa de Servicios Distribuidos') proporcionan muchas ventajas arquitecturales, pero a un alto coste de implementación. Los 'Conjuntos de Cambios de Registros' (DataSets, DataTables, etc.) no tienen a nivel de Arquitectura muchas ventajas, porque no cumplen el principio PI (*Persistance Ignorance*), no ofrecen alineamiento con Entidades lógicas de un modelo, etc., pero sin embargo son muy fáciles de implementar (p.e. los DataSet de ADO.NET son ideales para aplicaciones poco complejas a nivel de lógica de negocio y muy orientadas a datos, es decir, Arquitecturas nada orientadas al Dominio, 'nada DDD').

En definitiva, recomendamos una balance pragmático y ágil entre dichas preocupaciones (Productividad vs. Arquitectura Purista), **siendo la elección inicial (con la tecnología actual, EF 4.0) más acertada las STE (*Self-Tracking Entities*)** si se tiene un control extremo a extremo de la aplicación (Servidor y Cliente consumidor de los Servicios), pudiéndonos mover hacia los **DTOs** si la situación lo requiere (ofrecer nuestros servicios a consumidores desconocidos, o porque simplemente se

desea desacoplar el modelo de entidades del dominio del modelo de datos de la capa de presentación).

Las STE de Entity Framework 4.0 nos van a proporcionar una gran productividad y aun así consiguiendo beneficios arquitecturales muy importantes (Son entidades IPOCO, que cumplen en principio de PI, *Persistance Ignorance*) y desde luego representan un balance mucho mejor que los *DataSets* o las entidades simples/nativas de EF. Los DTOs, por otra parte, son definitivamente la mejor opción según una aplicación se hace más grande y más compleja o si tenemos requerimientos que no pueden cumplirse por las STE, como diferentes ratios/ritmos de cambios en el Dominio con respecto a la Capa de Presentación que hace deseable desacoplar las entidades del dominio de entidades/modelo de capa de presentación.

Estos dos patrones de implementación de objetos serializados a viajar entre niveles físicos (STE y DTO), son probablemente los más importantes a tener en cuenta en Arquitecturas **N-Tier** que al mismo tiempo sean Arquitecturas DDD (para aplicaciones complejas y orientadas al Dominio).

24.- CÓDIGO DE SERVICIO WCF PUBLICANDO LÓGICA DE APLICACIÓN Y DOMINIO

La publicación de lógica de Aplicación y Dominio normalmente no debe ser directa. Es decir, normalmente no daremos una visibilidad directa de las clases y métodos del Dominio o de la Capa de Aplicación. Por el contrario, deberíamos implementar un interfaz de Servicio Web que muestre lo que interesa ser consumido por el Cliente remoto (Capa de Presentación u otras aplicaciones externas). Por lo tanto, lo normal será realizar una granularización más gruesa, intentando minimizar el número de llamadas remotas desde la capa de Presentación.

24.1.-Desacoplamiento de objetos de capas internas de la Arquitectura, mediante UNITY

El uso de UNITY desde la Capa de Servicios Web, es crucial, pues es aquí, en los Servicios Web WCF donde tenemos el punto de entrada a los componentes del Servidor de Aplicaciones y por lo tanto es aquí donde debemos comenzar con el uso inicial explícito del contenedor de UNITY (Explícitamente determinando los objetos a instanciar, con Resolve()). En el resto de Capas (Aplicación, Dominio, Persistencia), se hace uso de UNITY también, pero automáticamente mediante la inyección de dependencias que utilizamos en los constructores. Pero la recomendación para una correcta y consistente 'Inyección de dependencias' es: '**solo deberemos hacer uso de "Resolve()" en el punto de entrada de nuestro Servidor (Servicios Web en una aplicación N-Tier, o código .NET de páginas ASP.NET en una aplicación Web)'**.

> **Uso Explícito de Contenedor de UNITY solo en punto de entrada al Servidor**: Solo deberemos hacer uso de "**Resolve()**" en el punto de entrada de nuestro Servidor (Bien sean Servicios Web en una aplicación N-Tier, o código .NET de páginas ASP.NET en una aplicación Web).

A continuación mostramos un ejemplo de clase de Servicio WCF que publica cierta lógica de la Capa de Aplicación:

```
C#
namespace Microsoft.Samples.NLayerApp.DistributedServices.MainModule
{
    public partial class MainModuleService
    {
```

Si controlamos las aplicaciones Cliente (Consumidores), devolvemos una Entidad POCO/IPOCO(STE) de EF 4.0. Si desconocemos quien consumirá el Servicio Web, debería ser mejor un DTO.

```
        public Customer GetCustomerByCode(string customerCode)
        {
            try
            {
```

RESOLUCION 'Raiz' con UNITY: Uso de Resolve<Interface>() de UNITY solo en puntos de entrada al Servidor, como Servicios WCF

```
                ICustomerManagementService      customerService      =
ServiceFactory.Current.Resolve<ICustomerManagementService>();
```

Llamada a un método de un Servicio de la Capa de Aplicación

```
                return customerService.FindCustomerByCode(customerCode);
            }
            catch (ArgumentNullException ex)
            {
                //Trace  data  for  internal  health  system  and  return
specific FaultException here!
                //Log  and  throw  is  a  known  anti-pattern  but  at  this
point (entry point for clients this is admitted!)

                //log exception for manage health system
                TraceManager.TraceError(ex.Message);

                //propagate exception to client
                ServiceError detailedError = new ServiceError()
                {
                    ErrorMessage                                         =
Resources.Messages.exception_InvalidArguments
                };
                throw new FaultException<ServiceError>(detailedError);
            }
        }
    }
}
```

24.2.-Gestión de Excepciones en Servicios WCF

Las excepciones internas que se produzcan, por defecto, WCF las transforma a FAULTS (para poder serializarlas y lleguen a la aplicación cliente consumidora). Sin embargo, a no ser que lo cambiemos, la información incluida sobre la Excepción dentro de la FAULT, es genérica (Un mensaje como "*The server was unable to process the request due to an internal error.*"). Es decir, por seguridad, no se incluye información sobre la Excepción, pues podríamos estar enviando información sensible relativa a un problema que se ha producido.

Pero a la hora de estar haciendo *Debugging* y ver qué error se ha producido, es necesario poder hacer que le llegue al cliente información concreta del error/excepción interna del servidor (por ejemplo, el error específico de un problema de acceso a Base de Datos, etc.). Para eso, debemos incluir la siguiente configuración en el fichero Web.config, indicando que queremos que se incluya la información de todas las excepciones en las FAULTS de WCF cuando el tipo de compilación es 'DEBUG':

CONFIG XML

```
<behavior name="CommonServiceBehavior">
                    ...
                    <serviceDebug includeExceptionDetailInFaults="true" />

                    ...
</behavior>
```

Incluir info de las Excepciones en las FAULTS, solo en modo DEBUGGING

En este caso, siempre que la compilación sea "*debug*", entonces si se incluirán los detalles de las excepciones en las FAULTS.

CONFIG XML

Estamos con compilación "debug" → Se mandará detalles de las excepciones

```
<system.web>
        <compilation debug="true" targetFramework="4.0" />
</system.web>
```

24.3.- Tipos de alojamiento de Servicios WCF y su implementación

El 'hosting' o alojamiento de un servicio WCF se puede realizar en diferentes procesos. Si bien puede ser en prácticamente cualquier tipo de proceso (.exe), inicialmente se pueden diferenciar dos tipos de alojamiento de servicios WCF:

- ***Self-Hosting***

   - El proceso (.exe), donde correrá nuestro servicio, es código nuestro, bien una aplicación de consola como hemos visto, o un servicio Windows, una aplicación de formularios, etc. En este escenario, nosotros somos los responsables de programar dicho alojamiento de WCF. A esto, en términos WCF, se le conoce como 'Self-Hosting' ó 'auto-alojamiento'.

- ***Hosting (IIS/WAS y Windows Server AppFabric)***

   - El proceso (.exe) , donde correrá nuestro servicio, está basado en IIS/WAS. IIS 6.0/5.5 si estamos en Windows Server 2003 ó Windows XP, o IIS 7.x y WAS solamente para versiones de IIS a partir de la versión 7.0.

La siguiente tabla muestra las características diferenciales de cada tipo principal:

Tabla 14.- Self-Hosting vs. IIS y AppFabric

	Self-Hosting	**IIS/WAS/AppFabric**
Proceso de alojamiento	Nuestro propio proceso, código fuente nuestro	Proceso de IIS/WAS. Se configura con una archivo/página .svc
Fichero de configuración	App.config	Web.config
Direccionamiento	El especificado en los EndPoints	Depende de la configuración de los directorios virtuales
Reciclado y monitorización automática	NO	SI

A continuación vamos a ir revisando los diferentes tipos específicos más habituales de *hosting*.

Hosting en Aplicación de Consola

Este tipo de alojamiento es precisamente como el que hemos visto anteriormente, por lo que no vamos a volver a explicarlo. Recalcar solamente que es útil para realizar pruebas, ***debugging***, demos, etc., pero no se debe de utilizar para desplegar servicios WCF en producción.

Hosting en Servicio Windows

Este tipo de alojamiento es el que utilizaríamos en un entorno de producción si no queremos/podemos basarnos en IIS. Por ejemplo, si el sistema operativo servidor es Windows Server 2003 (no disponemos por lo tanto de WAS) y además quisiéramos basarnos en un protocolo diferente a HTTP (por ejemplo queremos utilizar TCP, Named-Pipes ó MSMQ), entonces la opción que debemos utilizar para hosting es un servicio Windows desarrollado por nosotros (Servicio gestionados por el **Service Control Manager** de Windows, para arrancar/parar el servicio, etc.).

En definitiva se configura de forma muy parecida a un *hosting* de aplicación de consola (como el que hemos visto antes, app.config, etc.), pero varía donde debemos programar el código de arranque/creación de nuestro servicio, que sería similar al siguiente (código en un proyecto de tipo 'Servicio-Windows'):

```
namespace HostServicioWindows
{
    public partial class HostServicioWin : ServiceBase
    {
        //(CDLTLL) Host WCF
        ServiceHost _Host;

        public HostServicioWin()
        {
            InitializeComponent();
        }

        protected override void OnStart(string[] args)
        {
            Type tipoServicio = typeof(Saludo);

            _Host = new ServiceHost(tipoServicio);

            _Host.Open();

            EventLog.WriteEntry("Servicio Host-WCF", "Está disponible
el Servicio WCF.");
        }

        protected override void OnStop()
        {
            _Host.Close();
            EventLog.WriteEntry("Servicio Host-WCF", "Servicio WCF
parado.");
        }
    }
}
```

En definitiva tenemos que instanciar el servicio-wcf cuando arranca el servicio-windows (método OnStart()), y cerrar el servicio WCF en el método OnStop(). Por lo demás, el proceso es un servicio Windows típico desarrollado en .NET, en lo que no vamos a entrar en más detalles.

Hosting en IIS (Internet Information Server)

Es posible activar servicios de WCF utilizando IIS con técnicas de alojamiento similares a los tradicionales anteriores Servicios-Web (ASMX). Esto se implementa haciendo uso de ficheros con extensión .svc (comparable a los .asmx), dentro de los cuales se especifica en una línea el servicio que se quiere alojar:

```
<%@Service class="MiEmpresa.MiAplicacion.MiServicioWcf.Saludo" %>
```

Se sitúa este fichero en un directorio virtual y se despliega el *assembly* del servicio (.DLL) dentro de su directorio \bin o bien en el GAC (Global Assembly Cache). Cuando se utiliza esta técnica, tenemos que especificar el *endpoint* del servicio en el web.config, pero sin embargo, no hace falta especificar la dirección, puesto que está implícita en la localización del fichero .svc:

```
<configuration>
  <system.serviceModel>
    <services>
      <service type=" MiEmpresa.MiAplicacion.MiServicioWcf.Saludo ">
        <endpoint address=" binding="basicHttpBinding"
          contract=" MiEmpresa.MiAplicacion.MiServicioWcf.ISaludo "/>
      </service>
      ...
```

Si se navega con un *browser* al fichero .svc, se puede observar la página de ayuda para el consumo de dicho servicio, demostrándose que se ha creado automáticamente un ServiceHost dentro del dominio de aplicación de ASP.NET. Esto es realizado por un 'HTTP-Module' que filtra las peticiones entrantes de tipo .svc, y automáticamente construye y abre el ServiceHost apropiado cuando se necesita. Cuando se crea un web-site basado en esta plantilla de proyecto, automáticamente genera el fichero .svc junto con su correspondiente clase de implementación (se encuentra dentro de la carpeta *App_Code*). También nos proporciona una configuración de un *endpoint* por defecto, en el fichero *web.config*.

Hosting en IIS 7.x – WAS

En versiones 5.1 y 6.0 de IIS, el modelo de activación de WCF está ligado al '*pipeline de ASP.NET*', y por lo tanto, ligado a protocolo HTTP. Sin embargo, con **IIS 7.0** o superior, (versión existente a partir **Windows Vista** y **Windows Server 2008**), se introduce un mecanismo de activación independiente al protocolo de transporte, conocido como '*Windows Activation Service*' (WAS). Con WAS, se pueden publicar servicios .svc sobre cualquier protocolo de transporte, como TCP, Named-Pipes, MSMQ, etc. (al igual que en procesos '*Self-Hosting*').

Para poder utilizar un protocolo diferente (como TCP) basándonos en WAS, es necesario configurarlo primero en IIS, tanto a nivel del 'Web-Site' como a nivel del directorio virtual.

Añadir un binding a un site de IIS 7.0/WAS

Para añadir un nuevo binding a un site de IIS/WAS, podemos hacerlo bien con el interfaz gráfico de IIS 7.0 (en Windows Server 2008) o bien desde línea de comandos invocando comandos de IIS (en Windows Vista y en Windows Server 2008).

Por ejemplo, aquí muestro los *bindings* configurados en un Web Site de IIS 7.x:

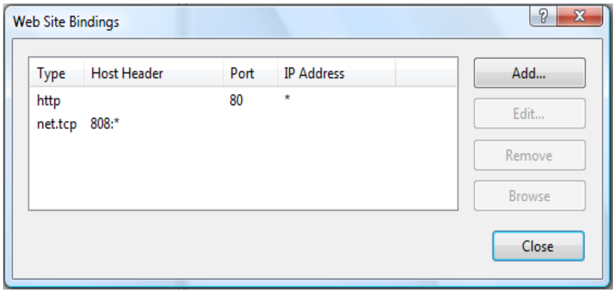


Figura 23.- Bindings en un Web Site de IIS 7.x

También puede añadirse con un *script* .cmd:

```
%windir%\system32\inetsrv\appcmd.exe set site "Default Web Site" -
+bindings.[protocol='net.tcp',bindingInformation='808:*']
```

Además debemos después habilitar los bindings que queramos poder utilizar en cada directorio virtual donde resida nuestro servicio wcf:

```
%windir%\system32\inetsrv\appcmd.exe set app "Default Web
Site/MiDirVirtual" /enabledProtocols:http,net.tcp
```

Hosting en Aplicación de Formularios Windows ó WPF

Puede parecer extraño que se quiera hacer *hosting* de un servicio (que aparentemente es un aspecto de un servidor), en una aplicación de formularios Windows o incluso en una aplicación de formularios WPF (Windows Presentation Foundation). Sin embargo, esto es tan raro, pues en WCF existe un tipo de binding '*Peer-to-Pee*r', donde las aplicaciones cliente son al mismo tiempo los servicios. Es decir, todas las aplicaciones hablan con todas porque son al mismo tiempo servicios WCF. En Internet hay muchos ejemplos de comunicación 'Peer-to-peer', como software de compartición de ficheros, etc.

La configuración en aplicaciones de formularios Windows ó WPF es similar a como lo hicimos en una aplicación de consola, por lo que tampoco en este caso vamos a entrar en detalle. En definitiva, es también otro tipo de '*Self-hosting*'. Lo que si cambia es la configuración interna en caso de querer hacer uso del *binding* 'Peer-to-Peer'.

Hosting en 'Windows Server AppFabric'

Similar al *hosting* en IIS 7.x (WAS), pero AppFabric aporta muchas capacidades extendidas de Monitorización de los servicios WCF y WF.

25.- DESPLIEGUE Y MONITORIZACIÓN DE SERVICIOS WCF EN WINDOWS SERVER APPFABRIC

Windows Server AppFabric tiene dos partes principales muy diferenciadas:

- **Hosting y monitorización de Servicios** (Servicios Web y Workflow Services), llamado en beta '*Dublin*'
- **Cache distribuido** (llamado en su beta '*Velocity*')

Relativo al cache de AppFabric, lo explicamos en el capítulo de '*Aspectos Transversales de la Arquitectura*'.

La parte de '**Hosting y monitorización de Servicios**' es precisamente relativa al despliegue de servicios web de WCF 4.0 y Workflow-Services de WF 4.0.

Lo que nos aporta como valor añadido esta parte son básicamente una serie de funcionalidades de facilidad de **despliegue** y sobre todo de **monitorización de los servicios WCF y WorkflowServices de WF**. Ahora, con AppFabric, un administrador de IT puede llegar a localizar problemas en servicios WCF de una aplicación (o validar su correcto funcionamiento) desde la consola de IIS 7.x. Antes de aparecer *Windows Server AppFabric*, la monitorización de los servicios WCF lo tenían que hacer los desarrolladores a mucho más bajo nivel (trazas, *debugging*, etc.) y apra un administrador de IT, los servicios WCF de una aplicación era 'una caja negra' de la cual desconocía su comportamiento.

El *Hosting* de AppFabric realmente no crea un entorno de hosting completamente nuevo. En lugar de eso se basa en IIS 7.x y WAS (*Windows Process Activation Service*) y añade ciertas capacidades de ejecución y gestión de servicios WCF incluyendo *Workflow-Services*, de forma similar a la figura siguiente.

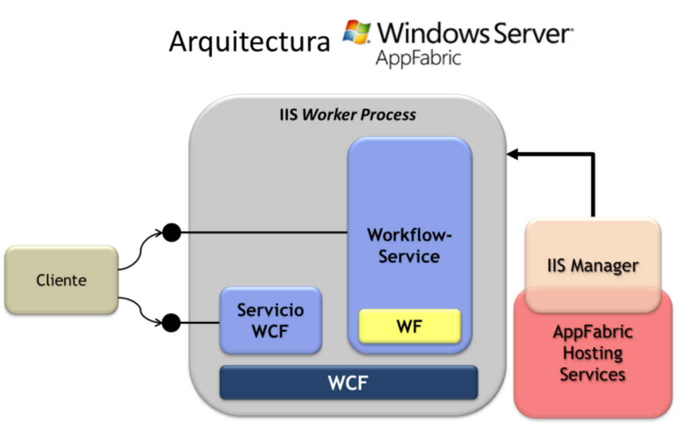


Figura 24.- 'Arquitectura de Windows Server AppFabric'

25.1.- Instalación y configuración de Windows Server AppFabric.

Lógicamente, lo primero que debemos realizar es instalar *Windows Server AppFabric,* bien directamente desde el *download* público de Microsoft, o bien mediante la plataforma '*Web Platform Installer*'. Una vez instalados 'los bits', debemos configurarlo con el asistente '*Windows Server AppFabric Configuration Wizard*', donde se configuran aspectos relativos al Cache y a la monitorización de Servicios WCF y WF. En este case vamos a revisar solamente lo necesario para la gestión y monitorización de Servicios y obviamos la parte del cache.

Primeramente, debemos configurar las bases de datos de monitorización a utilizar, así como las credenciales para realizar dicho acceso.

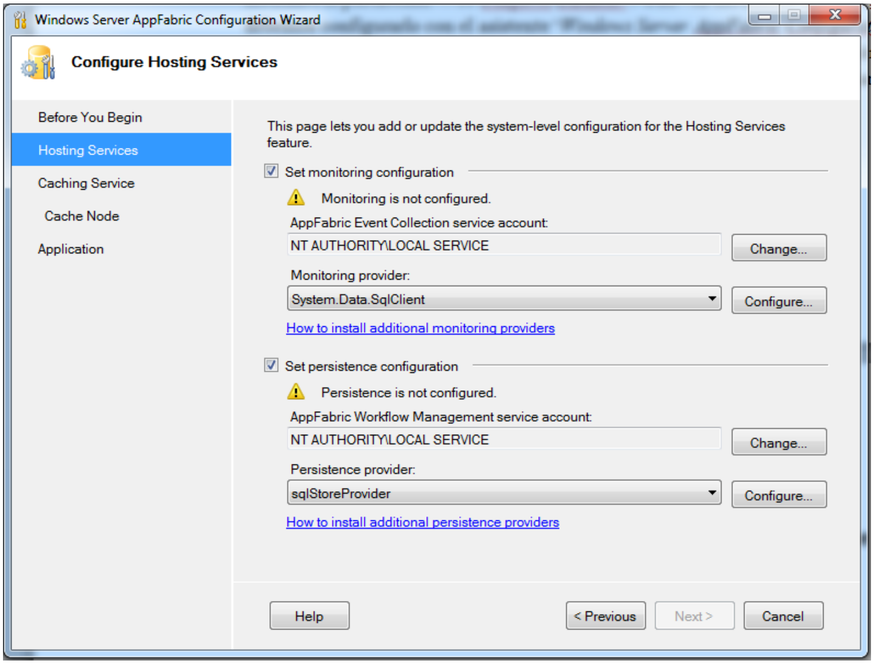


Figura 25.-

A nivel de hosting de servicios necesitaremos dos bases de datos de SQL Server. Una base de datos para la Monitorización de cualquier servicio WCF y Workflow-Services (llamada '*Monitoring Store*') y una segunda base de datos para la persistencia de la ejecución de workflows.

Para la base de datos de monitorización ('*Monitoring Store*'), en el ejemplo mostrado, hemos creado anteriormente (con *SQL Server Management Studio*) una base de datos vacía (llamada *AppFabricMonitoringDb*) en un SQL Server Express

local, pero perfectamente podría haberse utilizado un SQL Server Estándar remoto. Una vez disponible esa base de datos vacía, la especificamos en la configuración de AppFabric, como mostramos en la imagen.

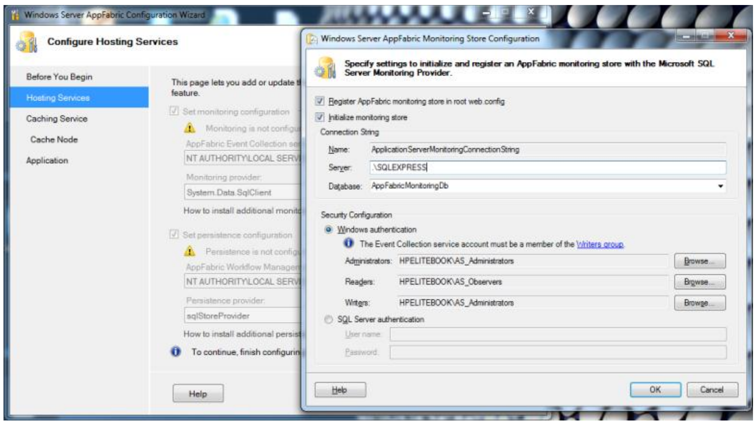


Figura 26.-

También puede optarse por hacer uso de autenticación SQL Estándar, con usuarios propios de SQL Server.

Y para la persistencia de ejecuciones largas de Workflow-Services, hemos creado otra base de datos, llamada por ejemplo '*AppFabricPersistenceDB*' y especificada también en la configuración de AppFabric.

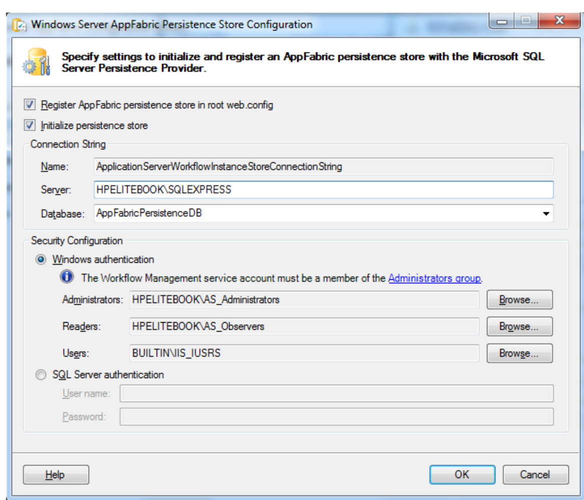


Figura 27.-

Quedando finalmente configurada la sección de Monitorización de AppFabric.

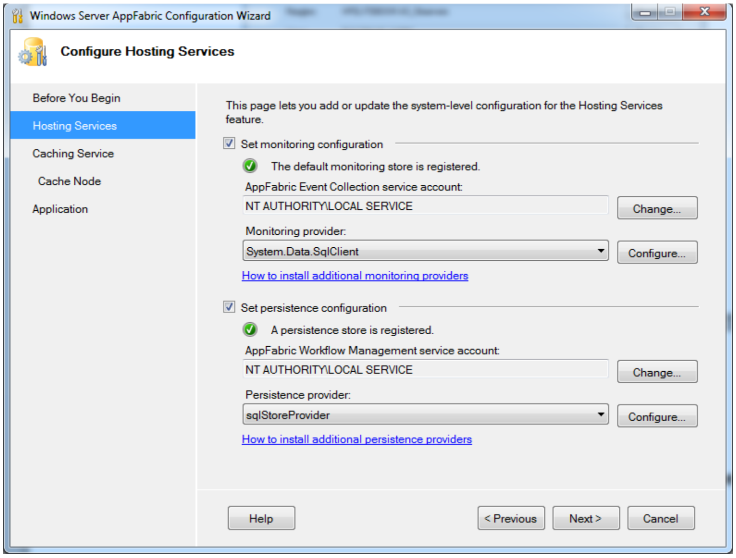


Figura 28.-

El resto de configuración del asistente lo omitimos porque es relativo a los servicios de Cache.

Si arrancamos IIS Manager, podremos observar unos nuevos iconos para administrar AppFabric:

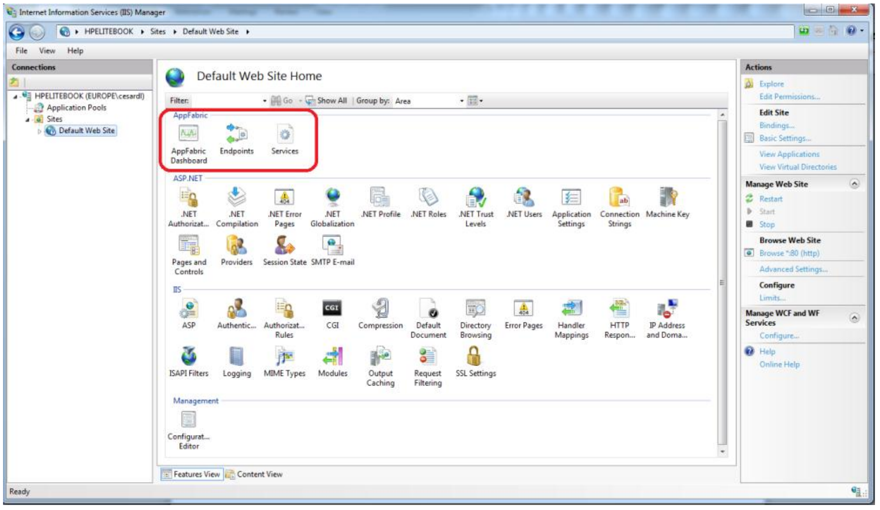


Figura 29.-

25.2.- Despliegue de servicios WCF en Windows Server AppFabric.

Una vez instalado y configurado *AppFabric*, podemos desplegar nuestros servicios WCF y componentes de aplicación/dominio en *AppFabric*, de forma que el uso de nuestros servicios WCF sean monitorizados.

En el entorno de desarrollo (nuestro PC), mapearemos directamente un web-site de IIS a nuestro proyecto de servicios WCF. Para ello primeramente debemos crear un Web-Site de IIS con el puerto TCP elegido para nuestros servicios WCF, por ejemplo el 8888 y especificamos como *path* raíz de dicho WebSite el directorio de nuestro proyecto de Servicios Distribuidos:

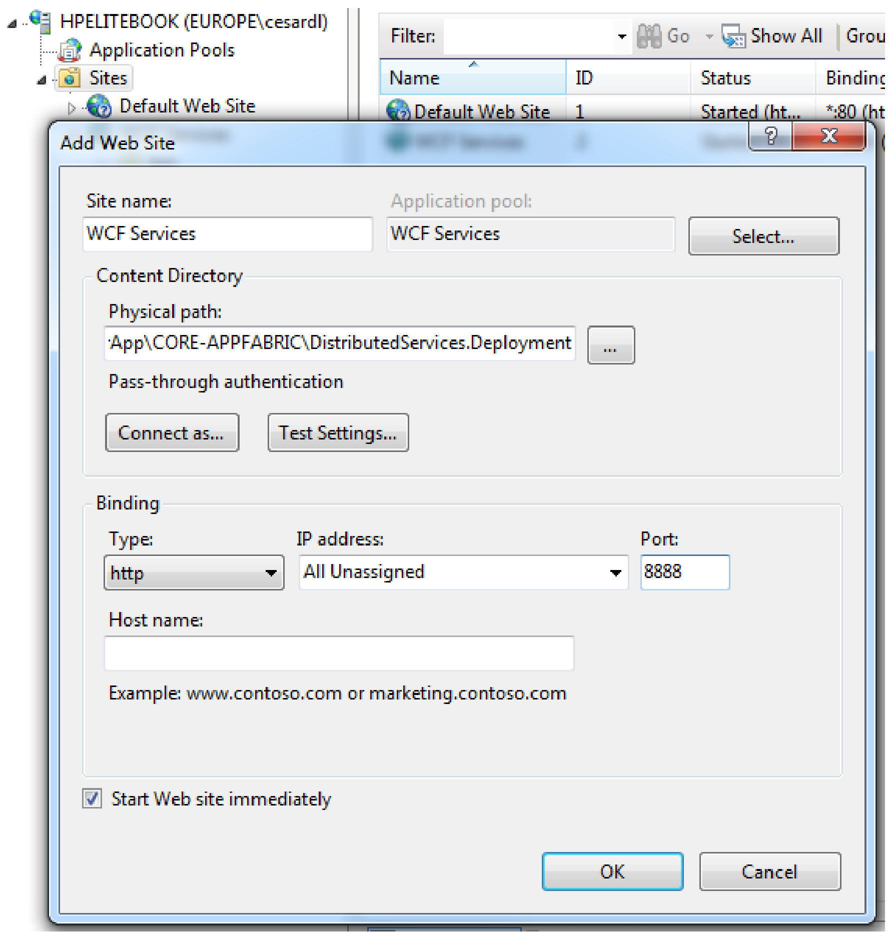


Figura 30.-

Con esto, deberemos de poder cambiar también las propiedades de nuestro proyecto WCF para que cuando hagamos *debugging* lo haga sobre IIS y no sobre 'cassini' de VS.2010. Necesitamos visualizamos en VS.2010 las propiedades del proyecto y en el tab 'Web' especificamos que queremos hacer uso del servidor Web IIS local. Como URL del proyecto especificaremos 'http://localhost:8888/' y finalmente debemos especificar en qué directorio físico se debe mapear el Directorio Virtual de IIS. Finalmente podemos también presionar el botón de '*Create Virtual Directory*' para comprobar que se conecta adecuadamente a IIS (aunque en este caso no crea ningún directorio virtual porque ya existe el root de nuestro WebSite).

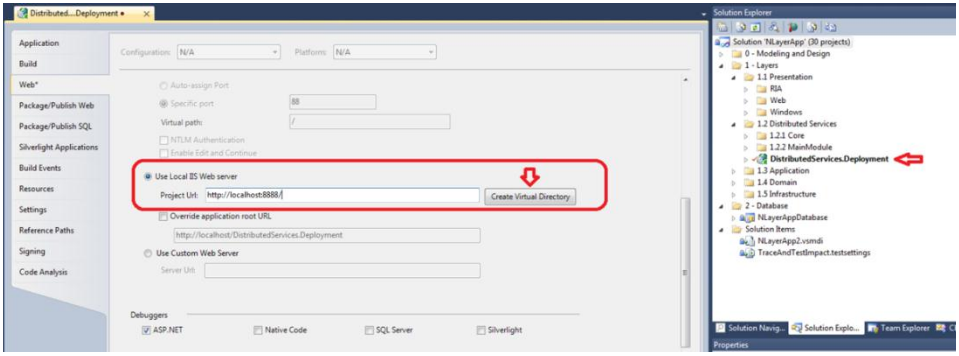


Figura 31.-

En este momento podemos llegar a probar la página de prueba de nuestro servicio WCF ejecutándose sobre IIS en el web-site con el puerto 8888 elegido. Bien lanzándolo desde el debugging de VS.2010 o también directamente desde el navegador o IIS Manager.

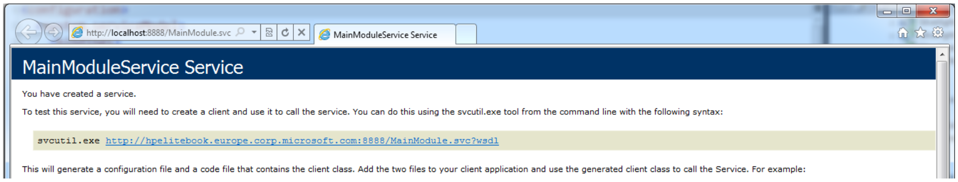


Figura 32.-

Por último, y para que los clientes de la aplicación funcionen correctamente, debemos cambiar la URL de la configuración de los *endpoints* WCF de consumo (en el XML del proyecto de Silverlight, WPF, etc.), especificando que el puerto ahora es el 8888 o cualquier otro cambio que hayamos realizado a la URL.

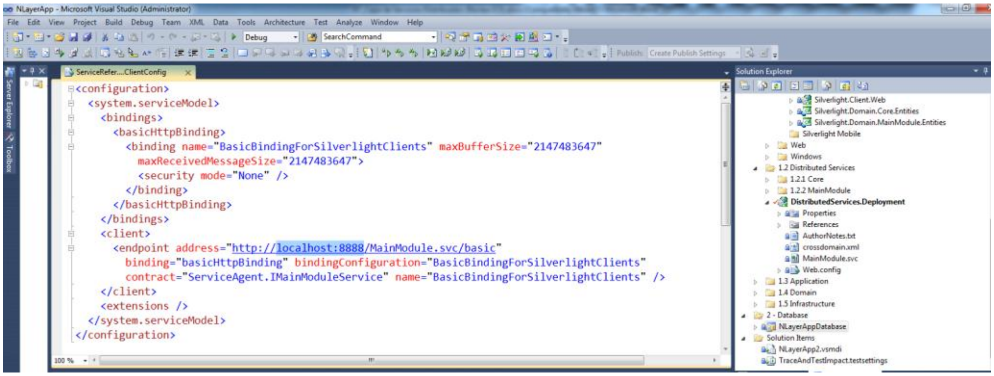


Figura 33.-

Finalmente, para desplegar el proyecto web/hosting de Silverlight también en IIS, simplemente necesitamos especificarlo en las propiedades del proyecto y crearlo como un directorio virtual de IIS, con el nombre del proyecto (puesto que el website para el puerto 80 ya existe y nuestro proyecto Silverlight no tiene por qué estar en el raíz de IIS). Presionando el botón '*Create Virtual Directory*' se creará el Directorio virtual en IIS.

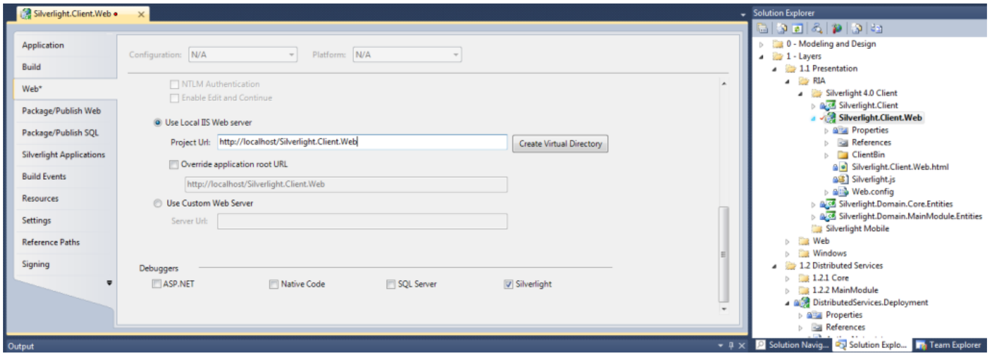


Figura 34.-

La misma operación puede realizarse con el proyecto de ASP.NET MVC para que se ejecute sobre IIS.

Finalmente la aplicación debería ejecutarse de forma similar, pero en este caso sobre IIS y AppFabric.

25.2.1.- Identidad de acceso a B.D. SQL Server e Impersonación de nuestra aplicación WCF

Hay que tener en cuenta que cuando estamos ejecutando nuestra aplicación en modo '*debugging*' desde Visual Studio y con servicios WCF basados en el '*Dev Web Server*' de Visual Studio (*Cassini*), y si la cadena de conexión utiliza 'Seguridad Integrada',

entonces la identidad del usuario que accede a SQL Server mediante las conexiones ADO.NET EF es la de nuestro usuario con el que hemos hecho login en la máquina, el cual probablemente no tiene problema alguno para acceder a nuestro SQL Server.

Por el contrario, si nuestro servicio WCF está ejecutándose en IIS/AppFabric, la identidad con la que intentará acceder a SQL Server será con la identidad del usuario con el que se ejecuta el 'pool' de procesos de nuestro Web-Site. Normalmente, 'ApplicationPoolIdentity' o 'NetworkService', los cuales probablemente no tendrán acceso a nuestro SQL Server.

Para que nuestros servicios WCF dispongan de acceso a SQL Server, normalmente se sigue el modelo 'Sistema Confiado' o '*Trusted Subsystem*'.

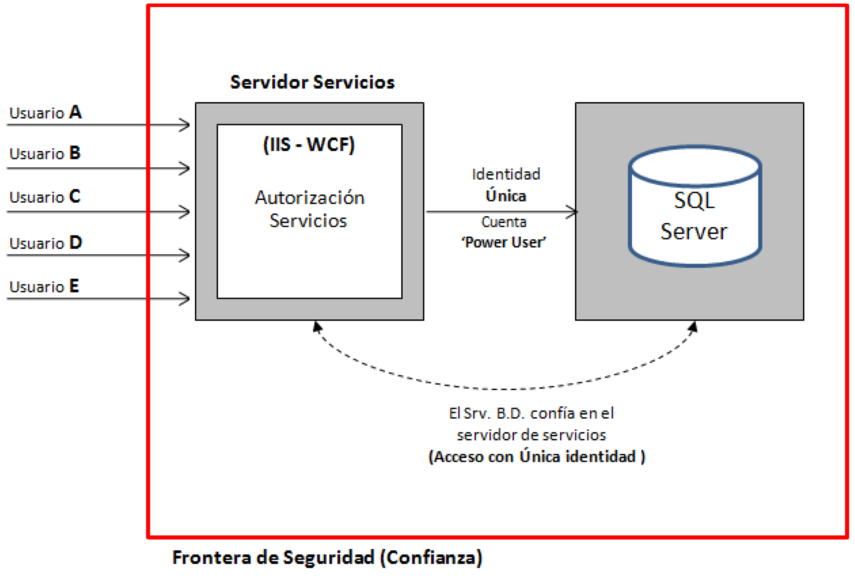
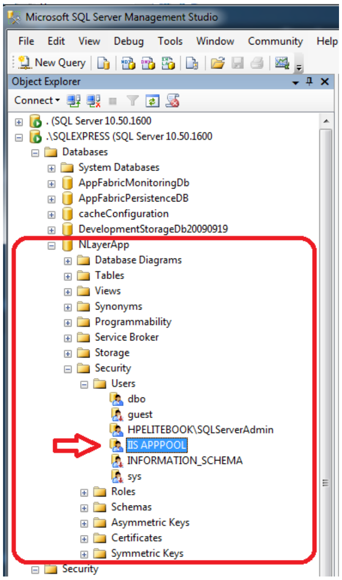


Figura 35.- Subsistema Confiado (Trusted Subsystem)

Teniendo una cadena de conexión EF/ADO.NET con modelo de seguridad Integrada, significa que necesitaremos que el Pool de IIS que utilizamos pueda acceder a nuestra base de datos en SQL Server, bien porque demos acceso a la B.D. al usuario con el que se ejecuta por defecto el Pool de IIS o porque cambiemos la identidad con la que se ejecuta el pool de IIS. Probablemente es más recomendable la primera opción, dar el acceso necesario hacia la B.D. al usuario por defecto del pool de IIS y así no tenemos que crear usuarios específicos nuevos de AD. Pero por supuesto, dependerá de nuestra aplicación y si se realizan otros accesos remotos con dicha identidad, etc.

Acceso a usuario del pool en SQL Server



Cambio de usuario del pool (Opcional)

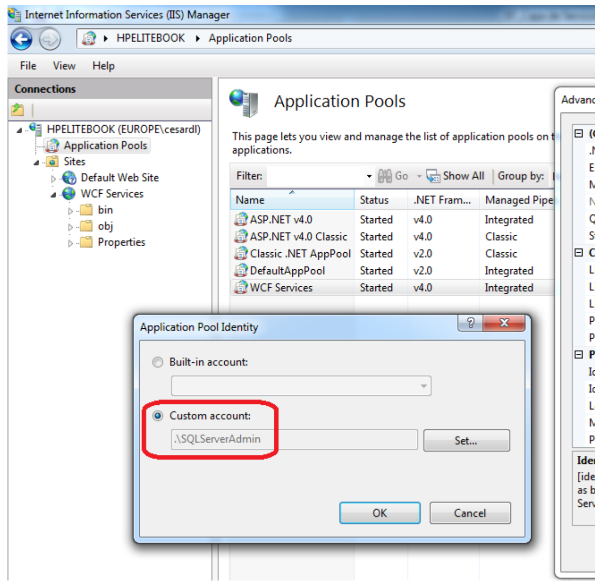


Figura 36.- Opciones de Identidad de acceso a B.D.

Otra opción es también, por supuesto, hacer uso de una cadena de conexión con seguridad estándar de SQL Server, donde las propias credenciales de acceso a SQL Server estarán en dicha cadena de conexión. Esta otra forma tiene la desventaja de ser menos seguro/recomendable pues tenemos que especificar en dicha cadena de conexión el usuario y password quedando estos datos visibles en nuestro fichero web.config. Normalmente al fichero web.config solo podrán acceder personal administrativo y desarrolladores, pero queda en cualquier caso menos seguro que especificar unas credenciales a nivel administrativo en la consola de IIS y quedando cifrado y protegido por el sistema de IIS (Dentro del fichero *applicationHost.config* quedan cifradas las credenciales a utilizar por el pool de IIS).

25.3.- Monitorización de servicios WCF desde la consola de Windows Server AppFabric en IIS Manager.

Habiendo realizado algunos accesos con la aplicación (desde Silverlight ó WPF) y por lo tanto, habiéndose consumido varias veces los servicios WCF de la aplicación desde que se instaló sobre AppFabric, podremos comprobar dichos accesos desde la consola de AppFabric.

Por ejemplo, desde el Dashboard, podemos tener una vista global.

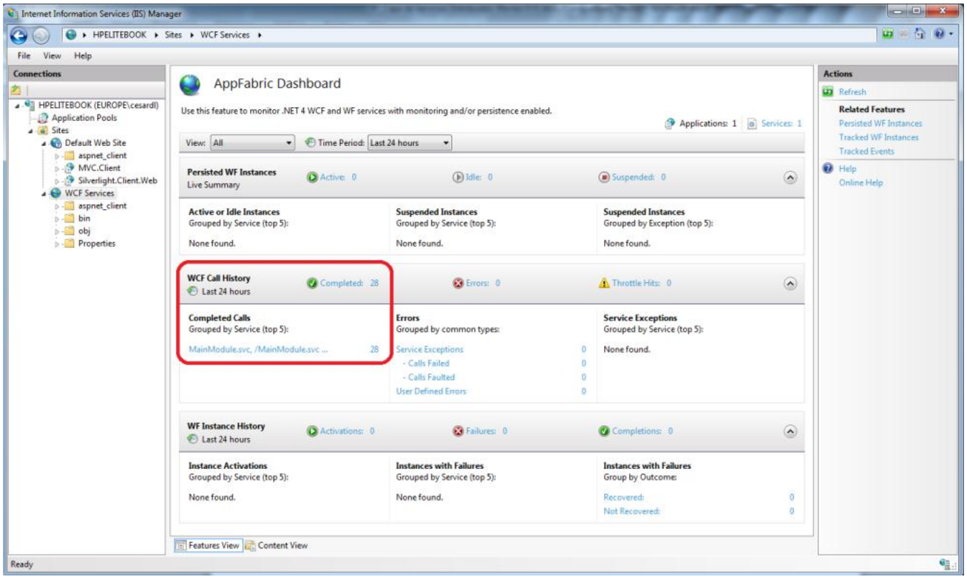


Figura 37.-

Y posteriormente podemos entrar en detalles de toda la lista de llamadas a servicios WCF. **Así pues, un administrador, sin ser desarrollador, puede examinar el comportamiento de los servicios web WCF de una aplicación, e incluso analizar los tiempos de ejecución de cada llamada a servicios web**, de forma que puede detectar problemas de rendimiento en una aplicación basada en Servicios Web aun no conociendo en detalle el desarrollo.

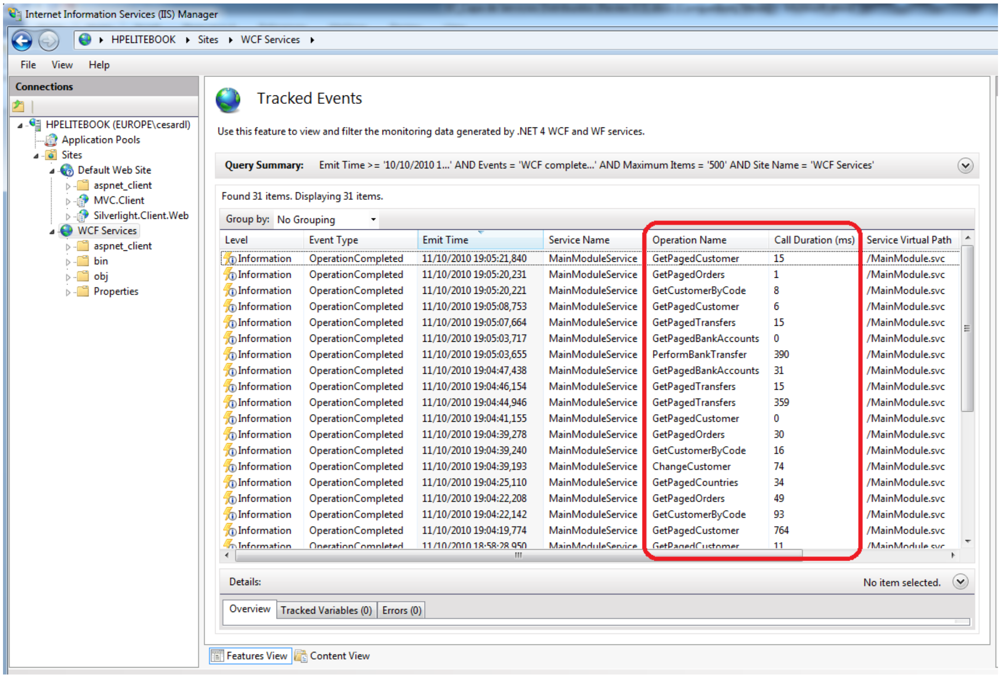


Figura 38.-

También podemos analizar las estadísticas globales para un servicio web concreto.

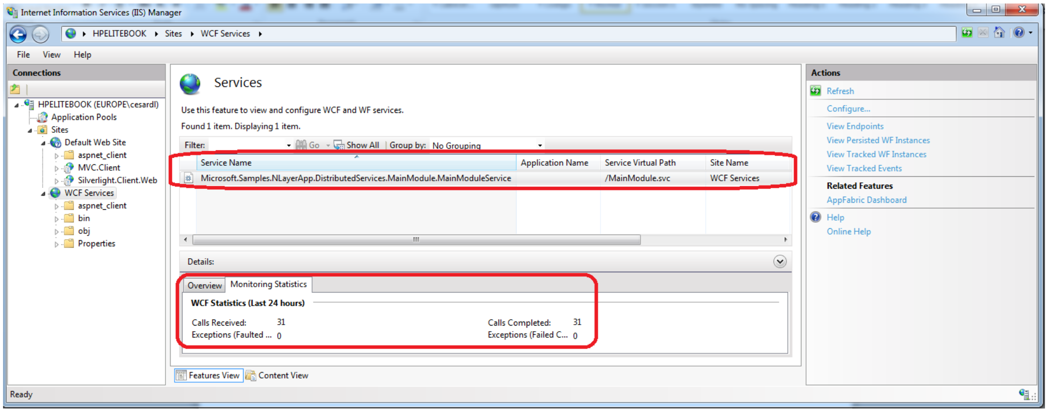


Figura 39.-

26.- REFERENCIAS GLOBALES DE WCF Y SERVICIOS

Exception Handling in Service Oriented Applications
http://msdn.microsoft.com/en-us/library/cc304819.aspx

"Data Transfer and Serialization":
http://msdn.microsoft.com/en-us/library/ms730035.aspx

"Endpoints: Addresses, Bindings, and Contracts":
http://msdn.microsoft.com/en-us/library/ms733107.aspx

"Messaging Patterns in Service-Oriented Architecture":
http://msdn.microsoft.com/en-us/library/aa480027.aspx

"Principles of Service Design: Service Versioning":
http://msdn.microsoft.com/en-us/library/ms954726.aspx

"Web Service Messaging with Web Services Enhancements 2.0":
http://msdn.microsoft.com/en-us/library/ms996948.aspx

"Web Services Protocols Interoperability Guide":
http://msdn.microsoft.com/en-us/library/ms734776.aspx

"Windows Communication Foundation Security":
http://msdn.microsoft.com/en-us/library/ms732362.aspx

"XML Web Services Using ASP.NET":
http://msdn.microsoft.com/en-us/library/ba0z6a33.aspx

"Enterprise Solution Patterns Using Microsoft .NET":
http://msdn.microsoft.com/en-us/library/ms998469.aspx

"Web Service Security Guidance":
http://msdn.microsoft.com/en-us/library/aa480545.aspx

"Improving Web Services Security: Scenarios and Implementation Guidance for WCF": **http://www.codeplex.com/WCFSecurityGuide**

"WS- Specifications":* **http://www.ws-standards.com/ws-atomictransaction.asp**

CAPÍTULO

8

Capa de Presentación

1.- SITUACIÓN EN ARQUITECTURA N-CAPAS

Esta sección describe el área de arquitectura relacionada con esta capa (relacionadas con la Interfaz de Usuario), donde presentaremos primeramente aspectos lógicos de diseño (patrones de diseño para la Capa de Presentación) y posteriormente y de forma diferenciada, la implementación de los diferentes patrones con diferentes tecnologías.

En el siguiente diagrama se muestra cómo encaja típicamente esta capa (Presentación), dentro de nuestra '*Arquitectura N-Capas Orientada al Dominio'*:

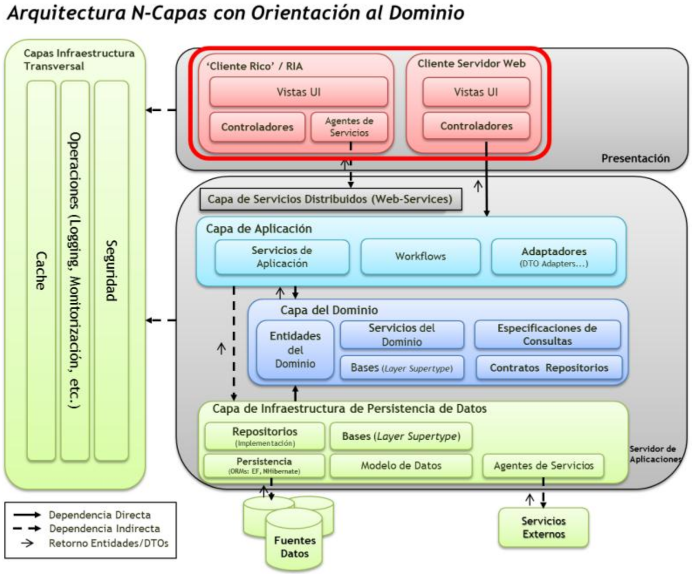


Figura 1.- Arquitectura N-Capas con Orientación al dominio

La Capa de Presentación incluye diferentes elementos como Vistas, Controladores, Modelos, etc. Sin embargo, antes de entrar en aspectos de Arquitectura, vamos a introducir aspectos importantes de la propia naturaleza de esta capa.

La responsabilidades de esta capa son las de presentar al usuario los conceptos de negocio mediante una interfaz de usuario (IU), facilitar la explotación de dichos procesos, informar sobre el estado de los mismos e implementar las reglas de validación de dicha interfaz. Al fin y al cabo, desde el punto de vista del usuario final, esta capa es la 'Aplicación' en sí, y de nada sirve el haber planteado la mejor arquitectura del mundo si no podemos facilitar la explotación de ella de la manera más satisfactoria posible para el usuario.

Una de las peculiaridades de las interfaces de usuario es que necesitan de unas habilidades que están fuera del ámbito del desarrollador, como pueden ser las habilidades de diseño artístico, los conocimientos de accesibilidad y de usabilidad, y el control de la localización de las aplicaciones. Por tanto, lo más recomendable es que un profesional de este ámbito como puede ser un diseñador gráfico, trabaje junto al desarrollador para lograr un resultado de alta calidad. Es responsabilidad de esta capa el facilitar esta colaboración entre ambos roles. El desarrollador programará en el lenguaje orientado a objetos elegido creando la lógica de la capa de presentación, y el diseñador usará otras herramientas y tecnologías como puede ser HTML o XAML para crear la parte visual y de interacción entre otras cosas.

Otras de las consideraciones en este apartado es que esta capa debe ser testeada al igual que las capas inferiores, y por tanto debe existir un mecanismo para automatizar dichos testeos y poderlo incluir en el proceso de integración automatizada, sea o no continua.

En la presente sección de la guía, sobre todo se quiere destacar el enfoque en dos niveles. Un primer nivel lógico (Arquitectura lógica), que podría ser implementado con cualquier tecnología y lenguajes (cualquier versión de .NET o incluso otras plataformas) y posteriormente un segundo nivel de implementación de tecnología, donde entraremos específicamente con versiones de tecnologías concretas.

2.- NECESIDADES DE INVERTIR EN LA INTERFAZ DE USUARIO

La inversión de las compañías en aplicaciones con interfaces de usuario intuitivas, simples y/o táctiles gracias al aumento de consumidores de este tipo de propuestas, ya sean mediante teléfonos móviles, mesas multi-táctiles o sistemas embebidos, las está dotando de ventajas comerciales. Se han escrito muchos estudios y se han impartido muchas presentaciones sobre este tema y todas llevan a las mismas conclusiones: al usar su software la productividad de usuario aumenta, los gastos se reducen y las ventas crecen.

Grandes empresas como Microsoft, Apple, Google, Yahoo o Amazon invierten mucho en la experiencia de usuario. Si se diseña una interfaz de usuario eficiente que permite y facilita a los usuarios solucionar tareas más rápidamente y en la mejor

manera, se logra un impacto enorme sobre la productividad de usuario. Tenga en cuenta que una interfaz de usuario bien diseñada y optimizada conlleva reducir el número de oportunidades de cometer errores por parte de los usuarios, lo que deriva a su vez en una mejora de la productividad. Si además de ser fácil e intuitiva, es rápida, sus usuarios serán capaces de más en menos tiempo; de nuevo hablamos de productividad. También entra en juego la psicología, si se crean interfaces de usuarios agradables, sin problemas de uso y sin fallos, los usuarios se sentirán más confortables a la hora de trabajar y por tanto más productivos.

En los días en los que estamos, la reducción de gastos es quizás la que más importe a las compañías. Si pensamos que tenemos una herramienta intuitiva, fácil de usar, la inversión en formación o documentación de la herramienta puede reducirse. Otra reducción la veremos cuando los usuarios se sientan cómodos usando las herramientas y tengan que usar menos el soporte. Otro factor importantísimo es usar la tecnología de UI adecuada para reducir complejidad y por tanto costes en el despliegue de la aplicación.

El punto más importante suele ser la diferenciación de mercado y la capacidad de ganar ventaja competitiva proporcionando a sus usuarios una experiencia de usuario superior a lo que ofrece el mercado actual. El dicho de que las primeras impresiones son las que venden, y una mala imagen puede hacer que un producto muy potente sea un estrepitoso fracaso.

Algunos estudios de terceros nos dan datos muy reveladores sobre la importancia de la interfaz de usuario. El estudio Gartner *"The Increasing Importance Of The Presentation Layer To The Enterprise"* publicado en 2004 ya apuntaba que *"La capa de presentación es integral a todos aspectos del proceso de desarrollo. Y debido a esto, vemos en todas partes que de un 25 a casi el 40 por ciento del tiempo total y del esfuerzo está de alguna manera relacionado con la capa de presentación en todas partes del proceso de desarrollo."*

Infragistics también tiene algunos números interesantes. Ellos citan que *"cada dólar invertido en UX trae entre 2 y 100 dólares a cambio"* y para demostrarlo se basan en:

1.- Gilb, T. (1988). Principios de Dirección de Ingeniería de Software. Addison Wesley

2.- Pressman, R.S. (1992). Ingeniería de Software. McGraw-Hill: Nueva York, Nueva York.

Un informe de SDS Consulting (Strategic Data Consulting) también ofrece algunos números interesantes. En su informe llamado *"Informe Especial: los Impactos de Negocio de UX y ROI"* ha concluido que las áreas de inversión en la experiencia de usuario son claves, y los resultados son la disminución en los gastos de desarrollo, el aumento de ingresos y la disminución del tiempo de salida al mercado. En dicho informe basado en la revisión de 735 compañías de Internet, concluyen que por término medio las empresas invierten el 11,5 % de sus presupuestos de desarrollo de un producto en interfaz de usuario. Y además que la interfaz de usuario está entre el 47% y el 66 % del código total de un proyecto, que causa el 40 % del esfuerzo de desarrollo

y que causa el 80 % de los apuros imprevistos. Lo interesante es que citan un caso real de McAfee donde cuentan que redujo el 90% de los gastos de soporte al modernizar su Interfaz de Usuario.

3.- NECESIDAD DE ARQUITECTURAS EN LA CAPA DE PRESENTACIÓN

Tratar la capa de presentación durante todas partes del ciclo de desarrollo (inclusive durante las etapas tempranas de la planificación de producto) reduce considerablemente el tiempo de desarrollo y los gastos económicos asociados a los cambios de diseño posteriores.

3.1.- Acoplamiento entre capas

El detalle de la separación entre código de la lógica de presentación y la interfaz amortigua los gastos ocasionados por cambios futuros, pero además favorece la colaboración entre diseñadores, desarrolladores y jefes de proyectos, lo que conlleva evitar pérdidas de tiempo por mala comunicación. Si la arquitectura está pensada para que el trabajo de un diseñador no interfiera en el trabajo de un desarrollador, conseguiremos reducir el tiempo de finalización del proyecto.

El principal problema que nos encontramos es el acoplamiento entre componentes y capas de la interfaz de usuario. Cuando una capa conoce cómo otra capa realiza su trabajo se dice que la aplicación está acoplada. Por ejemplo, el típico acoplamiento se da cuando en una aplicación se define una consulta de búsqueda y la lógica de esta búsqueda se programa en el controlador del botón "Buscar" que está en el código adjunto (*codebehind*). A medida que los requerimientos de la aplicación cambien y tengamos que ir modificándola, tendremos que actualizar el código de éste codebehind para la pantalla de búsqueda. Si por ejemplo hay un cambio en el modelo de datos tendremos que llevar esos cambios hasta la capa de presentación y comprobar que todo funciona de nuevo correctamente. Cuando todo está muy acoplado, cualquier cambio en una parte de la aplicación puede provocar cambios en el resto del código y esto presenta un problema de complejidad y calidad del código.

Cuando creamos una aplicación sencilla, como por ejemplo un reproductor de películas, este tipo de problemas no se suelen dar porque el tamaño de la aplicación es pequeño y el acoplamiento es aceptable y manejable. Pero en aplicaciones de Línea de Negocio (Business Line Applications o LOB), con cientos de pantallas o páginas se vuelve crítico. A medida que aumenta el tamaño de un proyecto éste se vuelve más complejo y tenemos más interfaces de usuario que mantener y comprender. Es por esto, que el patrón "Codebehind" se considera un anti-patrón en aplicaciones de línea de negocios.

3.2.- Búsqueda de rendimiento.

Por otra parte está el rendimiento. Muchas veces las optimizaciones consisten en rebuscados trucos que complican la comprensión del código. Estos trucos no suelen ser fáciles de entender y por tanto el mantenimiento de la aplicación se suele ver perjudicado. Evidentemente esto no es una verdad al 100%, pero si en la mayoría de los casos. Existen muchos trucos de rendimiento que para nada son complejos, y existen muchas aplicaciones con un alto rendimiento y muy mantenibles, pero nuestra experiencia nos ha demostrado que este tipo de situaciones suelen ser escasas. Así pues, en lo relativo a aplicaciones empresariales es preferible una buena mantenibilidad al máximo rendimiento gráfico, y por tanto debemos concentrarnos en optimizar sólo aquellas partes que son críticas para el sistema.

3.3.- Pruebas unitarias

Otra parte del problema son las pruebas unitarias. Cuando una aplicación está acoplada sólo se puede testear la parte funcional, es decir, la interfaz de usuario. De nuevo, esto no es un problema con un proyecto pequeño, pero a medida que un proyecto crece en tamaño y complejidad, poder comprobar por separado las capas de aplicación es crucial. Imagine que cambia algo en la capa de datos, ¿cómo comprueba que ha sido afectado en la capa de presentación? Tendría que realizar las pruebas funcionales de toda la aplicación, la realización de las pruebas funcionales es muy costoso debido a que no suelen estar automatizadas, requieren de personas que testeen una por una todas las funcionalidades. Es verdad que existen herramientas que sirven para grabar los movimientos de los usuarios y automatizar estas pruebas, pero la laboriosidad de realizarlas y el coste que ello implica hace que no se lleven a cabo.

Si además somos capaces de proporcionar al producto un conjunto de pruebas automatizadas mejoraremos la calidad de soporte postventa, ya que podremos reducir el tiempo en la localización y resolución de fallos (bugs). Con esto aumentaremos la calidad de nuestro producto, aumentaremos la satisfacción del cliente y reforzaremos su fidelidad.

Por estos motivos, se requiere que la capa de presentación tenga a su vez una sub arquitectura propia que encaje con el resto de la arquitectura propuesta.

4.- PATRONES DE ARQUITECTURA EN LA CAPA DE PRESENTACIÓN

Existen algunos patrones arquitectónicos conocidos que se pueden utilizar para el diseño de la arquitectura de la capa de presentación como pueden ser MVC, MVP, MVVM y otras variantes. A día de hoy, estos patrones se usan para separar los conceptos entre la interfaz de usuario y la lógica de presentación, pero en su momento fueron concebidos para separar la capa de presentación de la capa de negocio, ya que las tecnologías de presentación no solían abstraerse de lógica de negocio.

4.1.- Patrón MVC (Modelo Vista Controlador)

El patrón modelo-vista-controlador se remonta a los años ochenta, con su primera implementación en el popular lenguaje de programación SmallTalk. Este patrón ha sido uno de los más importantes e influyentes de la historia de la programación, tanto que hoy en día sigue siendo un patrón actual y ampliamente utilizado. Este patrón pertenece a un conjunto de patrones agrupados en el estilo arquitectural de presentación separada. El objetivo que buscamos al utilizar este patrón es separar el código responsable de la representación de los datos en pantalla, del código encargado de la ejecución de la lógica de negocio. Para ello el patrón divide la capa de presentación en tres tipos de objetos básicos: modelos, vistas y controladores. La utilidad del patrón reside en que describe los flujos de comunicación entre estos tres tipos de objetos, como muestra esta imagen:

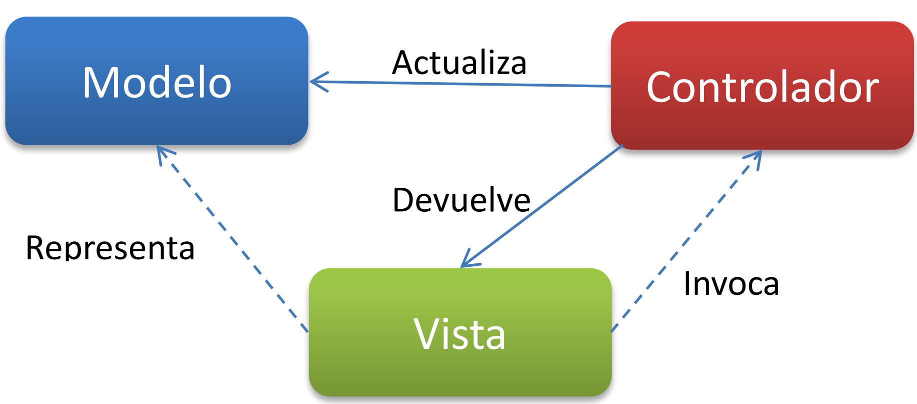


Figura 2.- Arquitectura MVC de la Capa de Presentación

En un escenario clásico la vista representa los datos del modelo e invoca acciones de un controlador en respuesta a las acciones del usuario. El controlador recoge las acciones del usuario, interactúa con el modelo y devuelve una determinada vista en respuesta.

En la mayor parte de las implementaciones de MVC, los tres componentes pueden relacionarse directamente el uno con el otro, pero en algunas implementaciones el controlador es responsable de determinar que vista mostrar. Esta es la evolución hacia el patrón Front Controller (**http://msdn.microsoft.com/en-us/library/ms978723.aspx**).

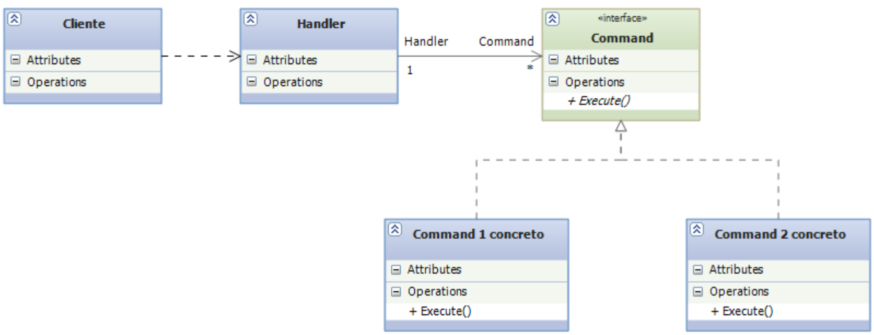


Figura 3.- Evolución Patron 'Front Controller'

En '*Front-Controller*', el controlador está divido en dos piezas, uno es el manejador (handler) que captura la información relevante de las peticiones que vienen desde la presentación y las dirige a un comando concreto, que es quien ejecuta la acción que se ha realizado en la presentación.

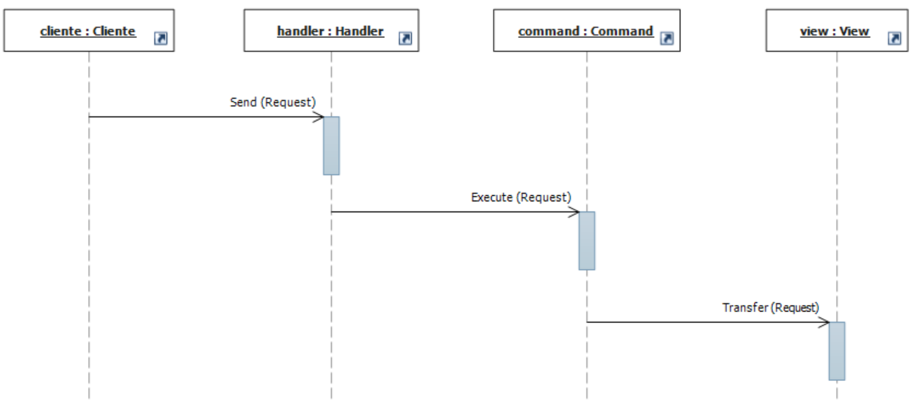


Figura 4.- Diagrama de Secuencia de Front-Controller

A continuación describiremos con más detalle las responsabilidades de cada uno de los componentes.

4.2.- El modelo

El modelo es el conjunto de clases encargado de representar la información con que trabaja el usuario. Es muy importante entender que estas clases pueden ser clases de dominio, DTOs o ViewModels (Entendiendo como viewmodel una clase que almacena los datos que representa una determinada vista, no un viewmodel como lo entendemos en Silverlight). Que opción escogemos depende de la situación:

- Clases de dominio: El modelo de la vista se corresponde fuertemente con una clase de dominio y no estamos empleando DTOs en la aplicación. Por ejemplo en una acción de editar una entidad.

- DTOs: El modelo de la vista se corresponde fuertemente con los datos de un DTO que estamos utilizando en nuestra aplicación.

- Viewmodels: Los datos que necesita la vista no se corresponden directamente ni con un DTO ni con una clase de dominio de nuestra aplicación. Este viewmodel almacena esa pequeña información adicional y probablemente incluye pequeños métodos de consulta (Propiedades) que simplifican la descripción de la vista. Por ejemplo: Un listado paginado de un conjunto de datos, donde el viewmodel se compone de los datos de la página y de propiedades como el número de página, su tamaño, el número total de páginas, etc.

4.3.- Las vistas

Las vistas son las encargadas de representar gráficamente el modelo y de ofrecer las acciones de los controladores para que el usuario pueda interactuar con este. Hay que tener muy claro que una vista no debe invocar ningún método que provoque un cambio de estado en el modelo o llamar a algún método que requiera parámetros, es decir, debe limitarse a acceder a propiedades simples y métodos de consulta del objeto que no tengan parámetros.

4.4.- El controlador

El controlador orquesta la interacción entre las vistas y el modelo. Recibe las peticiones del usuario, interactúa con el modelo realizando consultas y modificaciones a este, decide que vista se muestra en respuesta y le proporciona los datos requeridos para su renderizado, o delega la respuesta a otra acción de otro controlador.

4.5.- Patrón MVP (Modelo Vista Presentador)

En los años 90 se puso de moda el modelo "Formularios y Controladores" (*Forms & Controllers*) impulsado por entornos de desarrollo como Visual Basic o Delphi. La mayoria de estos entornos de desarrollo permitía que el desarrollador definiera la disposición de pantalla con un editor gráfico que permitía arrastrar y soltar controles dentro de un formulario. Con estos entornos, el número de aplicaciones creció exponencialmente debido a que facilitaba el desarrollo a la vez que se creaban interfaces de usuarios más agradables.

El formulario tiene principalmente dos responsabilidades:

- Disposición de pantalla: definición del posicionamiento de los controles en la pantalla, junto con la estructura jerárquica de unos controles con respecto a otros.

- Lógica del formulario: el comportamiento de los controles que suele responder a los eventos lanzados por los controles dispuestos por la pantalla.

La arquitectura propuesta por Forms & Controller se convirtió en el acercamiento dominante en el mundo de las arquitecturas de presentación. Este modelo proporciona un diseño que es fácil de entender y hace una separación buena entre componentes reutilizables y código específico de la lógica del formulario. De lo que carece Forms & Controller, y es primordial en MVC, es del patrón Presentación Separada y de la facilidad para programar un Modelo de Dominio.

En 1996, Mike Potel que era IBM Mainstream MVP pública el Modelo Vista Presentador, (Model View Presenter, MVP) que da un paso hacia la unión de estas corrientes tratando de tomar la mejor de cada una de ellas.

El patrón Model-View-Presenter (MVP) separa el modelo del dominio, la presentación y las acciones basadas en la interacción con el usuario en tres clases separadas. La vista le delega a su presenter toda la responsabilidad del manejo de los eventos del usuario. El presenter se encarga de actualizar el modelo cuando surge un evento en la vista, pero también es responsable de actualizar a la vista cuando el modelo le indica que ha cambiado. El modelo no conoce la existencia del presenter.

Por lo tanto, si el modelo cambia por acción de algún otro componente que no sea el presenter, debe disparar un evento para que el Presenter se entere.

A la hora de implementar este patrón, se identifican los siguientes componentes:

- IView: es la interfaz con la que el Presenter se comunica con la vista.
- View: vista que implementa la interfaz IView y se encarga de manejar los aspectos visuales. Mantiene una referencia a su Presenter al cual le delega la responsabilidad del manejo de los eventos.
- Presenter: contiene la lógica para responder a los eventos y manipula el estado de la vista mediante una referencia a la interfaz IView. El Presenter utiliza el modelo para saber cómo responder a los eventos. El presentador es responsable de establecer y administrar el estado de una vista.
- Model: Está compuesto por los objetos que conocen y manejan los datos dentro de la aplicación. Por ejemplo, pueden ser las clases que conforman el modelo del negocio (business entities).

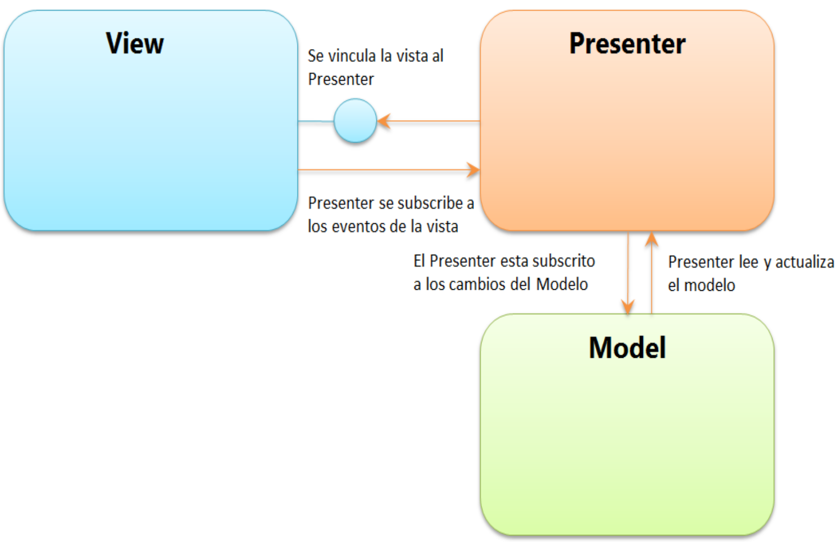


Figura 5.- Patrón MVP (Modelo Vista Presentador)

En julio de 2006, el archiconocido *Martin Fowler* publica una nota de retirada del patrón Presentation Model (como Fowler llamaba a MVP) dentro de su catálogo de patrones. Argumenta que tras un largo estudio y reflexión ha decidido dividir el patrón MVP en dos patrones dependiendo del nivel de responsabilidad de la vista:

- **Supervising Controller:** donde el controlador y la vista tienen funcionalidades repartidas, aunque específicas.

- **Passive View:** donde la vista es "controlada" por el controlador y tiene muy poca funcionalidad.

4.6.- Patrón MVVM (Model-View-ViewModel)

El patrón Model View ViewModel (MVVM) es también un derivado de MVP y este a su vez de MVC. Surgió como una implementación específica de estos patrones al hacer uso de ciertas capacidades muy potentes de nuevas tecnologías de interfaz de usuario disponibles. Concretamente a raíz de ciertas capacidades de Windows Presentation Foundation (WPF).

Este patrón fue adoptado y utilizado por el equipo de producto de Microsoft Expression desde la primera versión de Expression Blend desarrollado en WPF. Realmente, sin los aspectos específicos que aporta WPF y Silverlight, el patrón MVVM (Model-View-ViewModel) es prácticamente idéntico al PresentationModel definido por Martin Fowler, pero debido a las capacidades de las actuales tecnologías de interfaz de usuario, podemos exponerlo como un patrón genérico de arquitectura de capa de presentación.

El concepto fundamental de MVVM es separar el Modelo (Model) y la Vista (View) introduciendo una capa abstracta entre ellos, un "Modelo de la Vista" ó "ViewModel". La vista y el modelo de la Vista son instanciados normalmente por la aplicación contenedora. La vista guarda una referencia al ViewModel. El ViewModel expone comandos y entidades 'observables' o enlazables a los que la Vista puede enlazarse. Las interacciones del usuario con la Vista dispararán comandos contra el ViewModel y de forma análoga, las actualizaciones en el ViewModel se propagarán a la Vista de forma automática mediante enlace de datos.

El siguiente diagrama lo muestra a un alto nivel, sin entrar en detalles de implementación tecnológica:

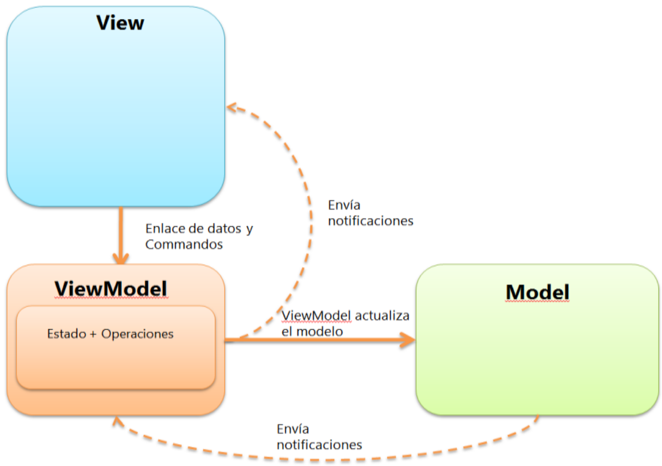


Figura 6.- Patrón MVVM (Model-View-ViewModel)

MVVM tiene como objetivo que los datos trasladados a la vista se puedan presentar y gestionar de la manera más sencilla. Es el ViewModel quien expone los datos desde el modelo y, en este sentido, el ViewModel es más un modelo que una vista. Pero además gestiona la lógica de visualización de la vista por lo que esta, desde este punto de vista, es más una vista que un modelo. A día de hoy, la mezcla de responsabilidades sigue siendo un tema de discusión y exploración continua en el sector.

4.7.- Visión global de MVVM en la arquitectura orientada a dominios

Dentro de la arquitectura de n-capas orientada al dominio, la sub arquitectura de la capa de presentación MVVM no está predispuesta en fila india como se puede llegar a pensar. El diagrama de abajo nuestra una visión de cómo se pueden comunicar las capas del MVVM con las capas de la arquitectura. Recordaros que cuando estamos hablando de capas no nos referimos a una separación física, sino a una separación lógica.

Fijaos cómo el Model puede estar o no comunicado con el resto de la arquitectura. El Model va a definir las entidades que se van a usar en la capa de presentación cuando ese modelo no sea igual al modelo de entidades. Si tenemos entidades que son iguales en la capa de dominio que en la de presentación, no vamos a repetir trabajo. El siguiente diagrama muestra que el ViewModel no tiene por qué comunicarse única y exclusivamente con el Model, sino que puede llamar directamente al resto de la

arquitectura orientada a dominios e incluso devolver entidades del dominio en vez del Model.

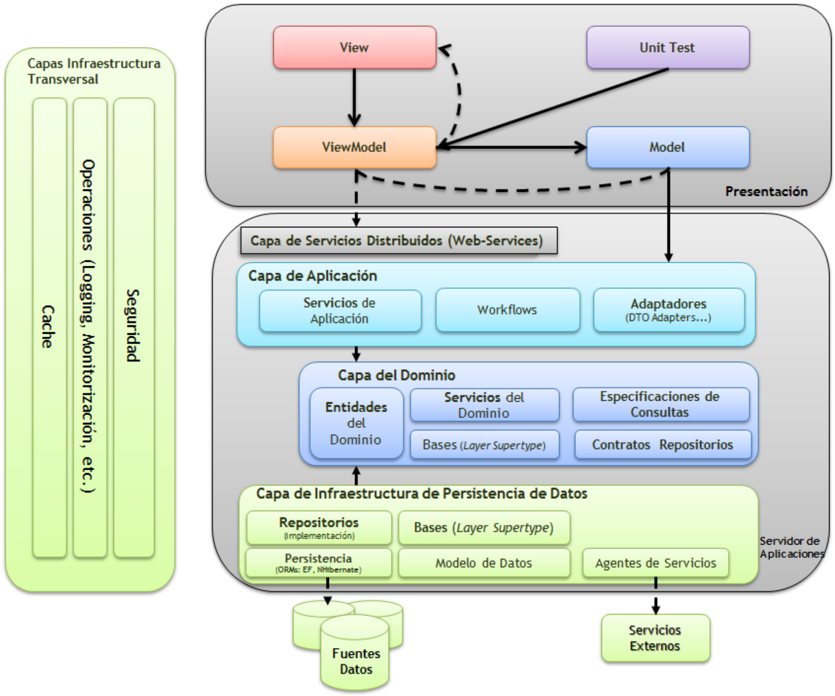


Figura 7.- Visión global de MVVM en la arquitectura DDD

4.8.- Patrones de diseño utilizados en MVVM

A continuación describiremos algunos patrones de diseño utilizado en el patrón arquitectónico MVVM.

4.8.1.-El patrón Comandos (Command)

El patrón de diseño Comandos (Command), también llamado Action o Transaction, tiene como objetivo aportar una interfaz abstracta de gestión de operaciones sobre un cierto receptor, permitiendo a un cliente ejecutar las operaciones sin tener que conocer exactamente el tipo de la operación ni quien la implementa.

Una de las motivaciones es que a veces se quiere poder enviar solicitudes a objetos sin conocer exactamente la operación solicitada ni el receptor de la solicitud. En general un objeto botón o menú ejecuta solicitudes pero la solicitud no está implementada dentro del mismo.

Se utiliza Command para:

- Parametrizar objetos por las acciones que realizan.

- Especificar, administrar y ejecutar solicitudes en tiempos distintos. El objeto Command tiene un tiempo de vida que es independiente de la solicitud del comando que lo instancia.

- Soporta la capacidad de deshacer la solicitud. el objeto Command puede guardar un estado que permita deshacer la ejecución del comando.

- Soporta la capacidad de generar históricos que permitan la recuperación del estado en caso de que el sistema falle.

- Permite estructurar un sistema en torno a operaciones de alto nivel construidas con base en operaciones primitivas o de bajo nivel.

La estructura es la siguiente:

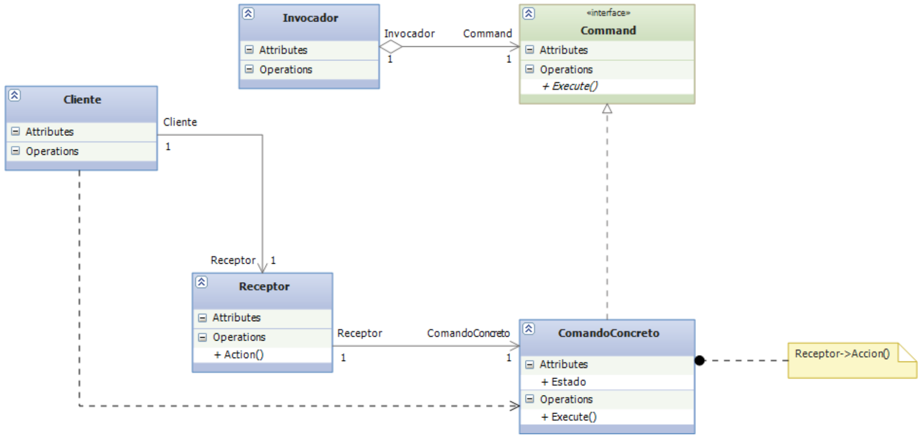


Figura 8.- Estructura Command

Donde las responsabilidades son:

- **Command** : Declara la interfaz para la ejecución de la operación.

- **ComandoConcreto :** Define la relación entre el objeto Receptor y una acción e implemeta Execute() al invocar las operaciones correspondientes en Receiver.

- **Cliente**: Crea un objeto ComandoConcreto y lo relaciona con su Receptor.
- **Invocador**: Le hace solicitudes al objeto *Command*
- **Receptor**: Sabe cómo ejecutar las operaciones asociadas a la solicitud. Cualquier clase puede ser receptora.

La colaboración entre objetos es la siguiente:

- Un cliente crea un objeto ComandoConcreto.
- El objeto Invocador guarda el objeto ComandoConcreto.
- El objeto Invocador solicita al llamar Execute() de *Command*.
- El objeto ComandoConcreto invoca las operaciones necesarias para resolver la solicitud.

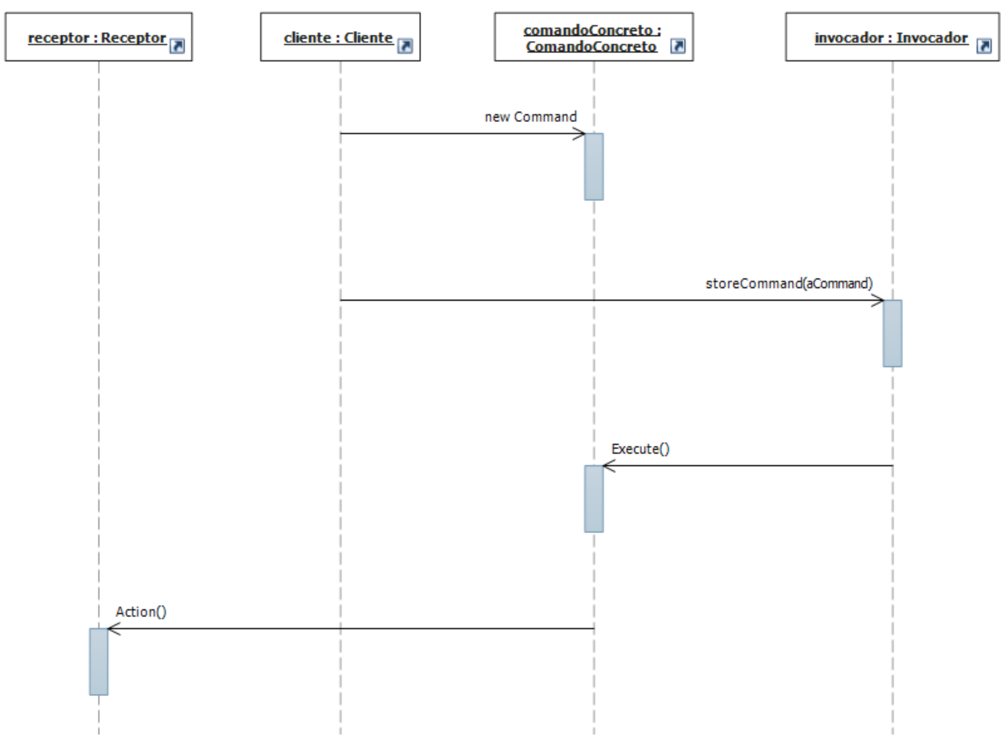


Figura 9.- Secuencia entre Objetos

4.8.2.-El patrón Observador (Observer)

El patrón Observador (Observer) o también conocido como el patrón publicación-subscripción, define una relación de un objeto a muchos objetos, de manera que cuando el objeto observable cambia su estado, se encarga de notificar este cambio a todos los observadores.

Las ideas básicas del patrón son sencillas: el objeto de datos contiene métodos mediante los cuales cualquier objeto observador se puede suscribir a él pasándole una referencia a sí mismo. El objeto de datos mantiene así una lista de las referencias a sus observadores.

Los observadores están obligados a implementar unos métodos determinados mediante los cuales el objeto de datos es capaz de notificar a sus observadores subscritos los cambios que sufre para que todos ellos tengan la oportunidad de refrescar el contenido representado.

Este patrón suele observarse en los frameworks de interfaces gráficas orientados a objetos, en los que la forma de capturar los eventos es suscribir a los objetos que pueden disparar eventos.

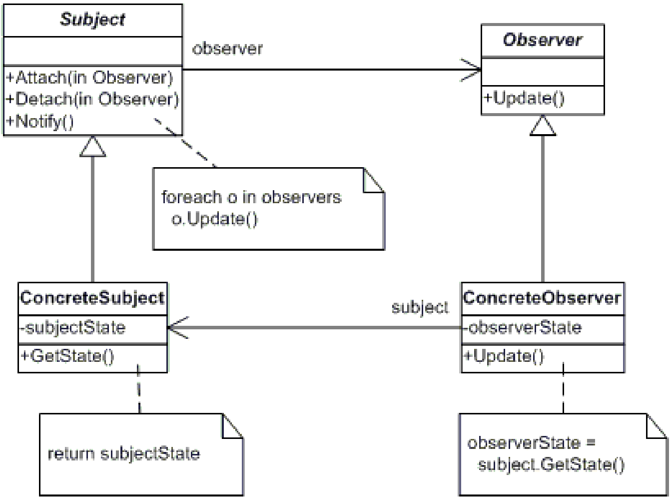


Figura 10.- El patrón Observador (Observer)

REFERENCIAS:

- *Presentation Model - Martin Fowler, July 2004.*

  http://martinfowler.com/eaaDev/PresentationModel.html

- *Design Patterns: Elements of Reusable Object-Oriented Software (ISBN 0-201-63361-2)*

  http://en.wikipedia.org/wiki/Special:BookSources/0201633612
  o Freeman, E; Sierra, K; Bates, B (2004). Head First Design Patterns. O'Reilly.

- *Introduction to Model/View/ViewModel pattern for building WPF apps - John Gossman, October 2005.*

  http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx

- *Separated Presentation - Martin Fowler, June 2006.*

  http://www.martinfowler.com/eaaDev/SeparatedPresentation.html

- *WPF Patterns - Bryan Likes, September 2006*

  http://blogs.sqlxml.org/bryantlikes/archive/2006/09/27/WPF-Patterns.aspx.

- *WPF patterns: MVC, MVP or MVVM or…? - the Orbifold, December 2006.*

  http://www.orbifold.net/default/?p=550

- *Model-see, Model-do, and the Poo is Optional - Mike Hillberg, May 2008.*

  http://blogs.msdn.com/mikehillberg/archive/2008/05/21/Model-see_2C00_-model-do.aspx

- *PRISM: Patterns for Building Composite Applications with WPF - Glenn Block, September 2008.*

  http://msdn.microsoft.com/en-us/magazine/cc785479.aspx

- *The ViewModel Pattern - David Hill, January 2009.*

  http://blogs.msdn.com/dphill/archive/2009/01/31/the-viewmodel-pattern.aspx

- *WPF Apps with the Model-View-ViewModel Design Pattern - Josh Smith, February 2009.*

  http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

- *Model View Presenter – Jean-Paul Boodhoo, August 2006.*

  http://msdn.microsoft.com/en-us/magazine/cc188690.aspx

5.- IMPLEMENTACIÓN DE CAPA DE PRESENTACIÓN

El objetivo del presente capítulo es mostrar las diferentes opciones que tenemos a nivel de tecnología para implementar la 'Capa de Presentación', dependiendo de la naturaleza de cada aplicación y patrones de diseño elegidos.

En el siguiente diagrama de Arquitectura resaltamos la situación de la Capa de Presentación:

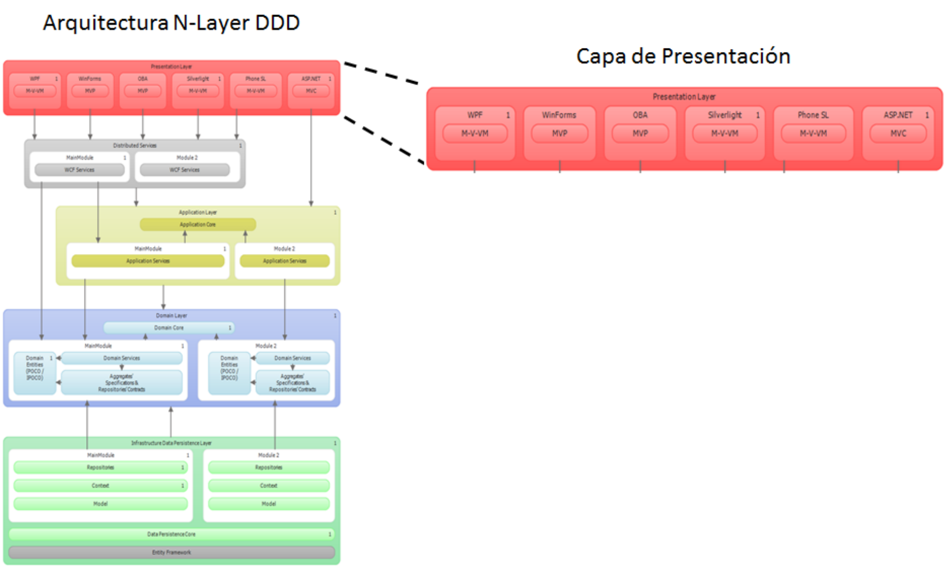


Figura 11.- Situación de Capa de Presentación en Diagrama Layer con VS.2010

Como vemos en detalle de la Capa de Presentación, en las aplicaciones de hoy nos encontramos distintas tecnologías y, además, interfaces de usuario que posibilitan usar distintas funcionalidades de distinta forma. Por ejemplo, en **Microsoft Office** podemos

cambiar el formato de un texto desde la barra de menús o desde el menú secundario asociado al botón derecho del ratón.

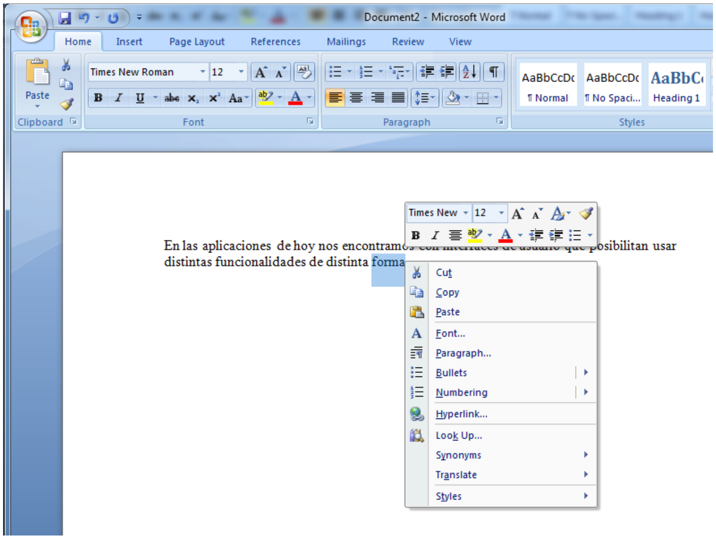


Figura 12.- Microsoft Office Word como Capa de Presentación

Si simplificamos el problema y queremos implementar una interfaz con funcionalidades repetidas desde múltiples sitios podemos encontrarnos con que en XAML podemos definir manejador de eventos tal que así:

```
    <Button    Name="UnoBtn"    Click="UnoBtn_Click"    Width="100"
 Height="25">Botón</Button>
```

Y también...

```
    Function test()
    <my1:Label KeyDown="Label_KeyDown">Uno</my1:Label>
```

Y luego programarlo desde el ***codebehind***:

```
    private void UnoHl_Click(object sender, RoutedEventArgs e)
            {
                Funcionalidad();
            }
            private void Label_KeyDown(object sender, KeyEventArgs e)
            {
                Funcionalidad();
            }
```

Debemos fijarnos en un detalle. Por una parte estamos definiendo el manejador de eventos desde el propio XAML y estamos mezclando dos roles. El diseñador, que

genera XAML de forma visual con Blend, no tiene por qué saber nada de cuál es el manejador. El entiende de conceptos o de órdenes. Por ejemplo, él sabe que el botón que está diseñando sirve para guardar y que por la tanto estará vinculado a la orden de Guardar.

Por otra parte, estamos creando una fuerte dependencia del XAML con la programación de la funcionalidad. Esto quiere decir que si queremos testear que se guarda algo, tendremos que testear el codebehind. Si esta funcionalidad la vamos a reutilizar en otros XAMLs tendremos que sacarlo a una clase distinta. Pero además seguimos obligando a que el programador tenga que ver los XAML y vincular los manejadores de eventos a cada XAML. Luego no hay total independencia de responsabilidades, o bien el diseñador entiende algo de programación porque tiene que vincular eventos o bien el programador tiene que vincular los eventos con lo que ha hecho el diseñador.

En el ejemplo anterior, por una acción realizada desde dos componentes XAML distintos, tenemos que usar dos manejadores de eventos, si tenemos la mala suerte de que estos manejadores tienen delegados distintos. Si nos ponemos en 10 funcionalidades repetidas desde tres componentes distintos, nos ponemos en 30 manejadores de eventos. ¿Parece inmanejable verdad? Por ello es recomendable utilizar una arquitectura de ayude a solventar estos problemas.

5.1.- Arquetipos, Tecnologías UX y Patrones de Diseño relacionados

Existen actualmente una serie de Arquetipos de aplicaciones determinados y marcados por la naturaleza de su tecnología visual, su Experiencia de Usuario y tecnología con la que están implementados. Estos Arquetipos están explicados uno a uno también en la presente guía de Arquitectura (incluyendo otros Arquetipos no marcados por tecnologías UX), si bien, de cara al estudio detallado de patrones y tecnologías de Capa de Presentación, solo vamos a revisar algunos de ellos.

La lista de Arquetipos actual definidos por aspecto UX (Experiencia de Usuario) sería:

- Aplicaciones **Cliente Rico** (Aplicaciones de Escritorio / Windows)
- Aplicaciones **Web** (Aplicaciones Dinámicas HTML)
- Aplicaciones **RIA** (*Rich Internet Applications*)
- Aplicaciones **Móviles**
- Aplicaciones **OBA** (*Office Business Applications*)

Dependiendo de cada Arquetipo, disponemos de una o más tecnologías para implementar las aplicaciones y a su vez, dependiendo de cada tecnología, se recomienda el diseño e implementación de uno u otro patrón de arquitectura de la Capa de Presentación. A continuación mostramos una matriz donde exponemos estas posibilidades:

Tabla 1.- Arquetipos y Patrones de Arquitectura de Capa de Presentación

Arquetipo	Tecnología	Patrón de Arquitectura – Capa Presentación
Aplicaciones Ricas (Aplicaciones de Escritorio / Windows)	• **WPF (*)** • WinForms	→ **MVVM** → MVP
Aplicaciones Web	• **ASP.NET MVC (*)** • ASP.NET Forms	→ **MVC** → MVP
Aplicaciones RIA	• **Silverlight (*)**	→ **MVVM**
Aplicaciones Móviles – Ricas	• .NET Compact Framework	→ MVP
Aplicaciones Móviles – RIA	• Silverlight Mobile	→ MVVM
Aplicaciones OBA (Office)	• .NET VSTO	→ MVP

En la presente edición inicial de esta Guía de Arquitectura, vamos a hacer hincapié específicamente en los Arquetipos y patrones relacionados con las tecnologías resaltadas en negrita y un (*), es decir:

- WPF
- Silverlight
- ASP.NET MVC

5.2.- Implementación de Patrón MVVM con WPF 4.0

La comunidad relacionada inicialmente con **WPF** ha creado un patrón denominado Model-View-ViewModel (MVVM). Este modelo es una adaptación de los patrones MVC y MVP en el que el ViewModel proporciona el modelo de datos y el comportamiento de la vista, pero permite a la vista enlazarlos mediante declaración en el modelo de vista. La vista se convierte en una combinación de XAML y C#, el modelo representa los datos disponibles para la aplicación y el modelo de vista prepara el modelo para ser enlazado por desde la vista.

MVVM fue diseñado para hacer uso de funciones específicas de WPF (disponibles también ahora a partir de Silverlight 4.0), que facilitan mucho mejor la separación de desarrollo/diseño entre la sub-capa de Vistas del resto del patrón, eliminando virtualmente todo el "code behind" (código adjunto en C#, VB, etc.) de la sub-capa de Vistas. En lugar de requerir Diseñadores Gráficos que escriban código .NET, los diseñadores gráficos pueden hacer uso del lenguaje de marcas XAML (Un formato específico basado en XML) de WPF para crear enlaces al ViewModel (El cual si es código .NET mantenido por desarrolladores). Esta separación de roles permite a los Diseñadores Gráficos centrarse en el UX (La Experiencia de Usuario) en lugar de tener que conocer algo de programación o lógica de negocio, permitiéndose finalmente que las diferentes sub-capas de Presentación sean creadas por diferentes equipos y de diferente perfil técnico.

El siguiente esquema representa este funcionamiento:

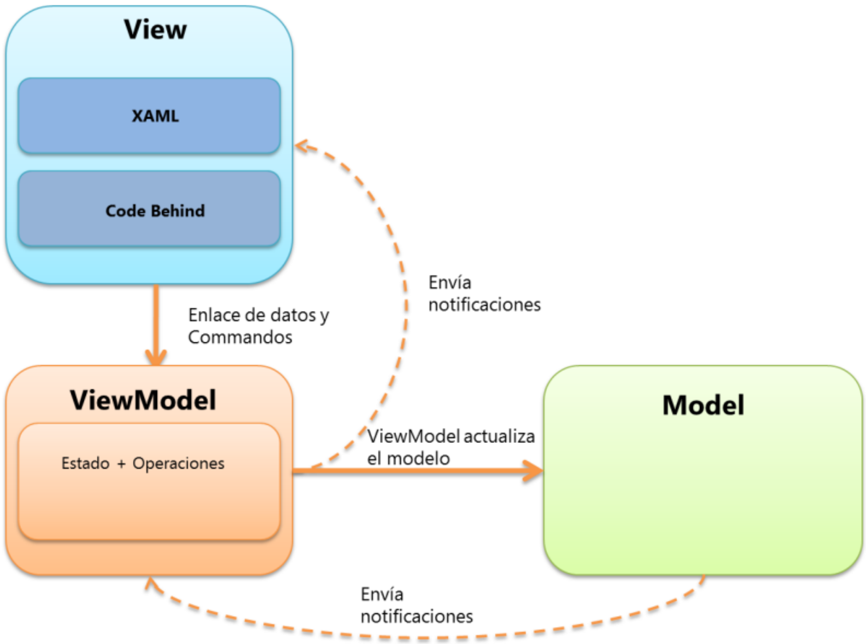


Figura 13.- Esquema

5.2.1.-Justificación de MVVM

Diseño Simplista de Aplicación

Para demostrar los beneficios del uso de una arquitectura que separa la presentación del modelo, vamos a comenzar con un escenario simple de una aplicación donde usamos el patrón *Separated Presentation*.

Supongamos un escenario donde tenemos dos ventanas (Vistas): Una Vista con una tabla/lista donde podemos ver todos los clientes de la aplicación, y otra vista mostrando los detalles de un solo cliente (de un solo ‘registro’ si lo simplificamos). Tendríamos un Interfaz de Usuario similar al siguiente:

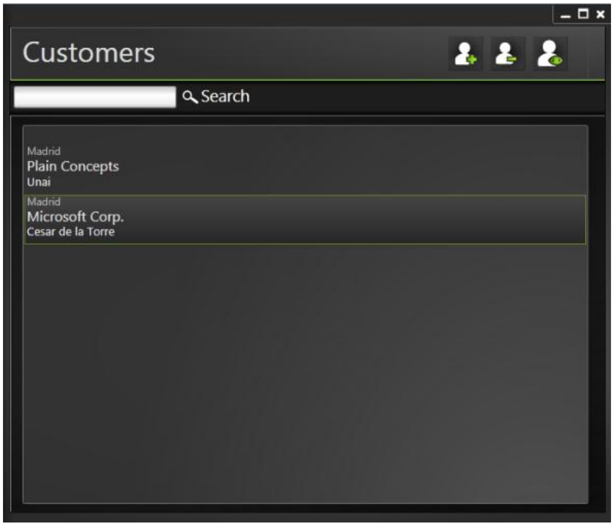


Figura 14.- Vista de Lista de Clientes



Figura 15.- Vista de Detalle de Cliente

En la separación entre Modelo y Vista para este escenario concreto, el Modelo serán los datos de los Clientes y las Vistas serán las dos Ventanas mostradas.

Siguiendo nuestro modelo simplista el diseño sería algo así:

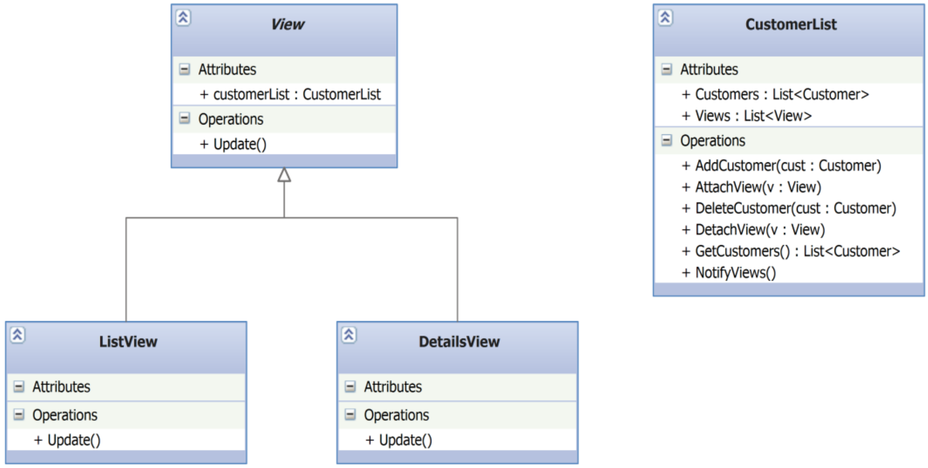


Figura 16.- Diseño Simplista – Las vistas están directamente conectadas al Modelo

Como se puede ver, **CustomerList** guarda una lista de vistas (que se podrían añadir o quitar con **AttachView**() y **DetachView**()). Cuando un contacto cambia, **CustomerList** notificaría a todas las Vistas llamando al método **Update**() y todas las vistas se actualizarían a sí mismas llamando al método del modelo **GetCustomers()**. Una vez instanciados, las vistas ListView y DetailsView dispondrán de una referencia a CustomerList, que lo guardan como un campo miembro (Definido en la clase base abstracta 'View', como customerList). Como ya se habrán dado cuenta se ha aplicado el patrón 'observer'.

Nota:
Destacar que las dos Vistas en este caso están FUERTEMENTE ACOPLADAS con la clase **CustomerList**. Si añadimos más lógica de negocio cliente, solo incrementará el nivel de acoplamiento.

Versión WPF del Diseño Simplista de Aplicación

Antes de continuar analizando el diseño lógico, vamos a convertirlo a un diseño amigable con WPF. Hay que tener en cuenta que WPF nos proporciona dos aspectos muy potentes que podemos utilizar directamente:

- **Databinding**: La capacidad de enlazar elementos de Interfaz Gráfico a cualquier dato.

- **Commands**: Nos proporcionan la capacidad de notificar, a los datos subyacentes, que se han producido cambios en el Interfaz Gráfico.

Según estas capacidades, el diseño amigable con WPF quedaría de la siguiente forma:

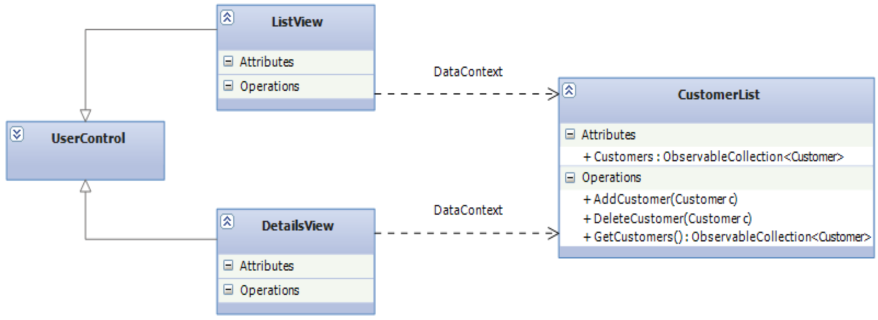


Figura 17.- Diseño Simplista y amigable para WPF de aplicación

Las vistas en este caso ya derivan de **UserControl** (no tenemos necesidad de una clase abstracta **View**) y **CustomerList** ya no necesita mantener una lista de Vistas, puesto que no le hace falta 'conocer' nada sobre las vistas. En lugar de eso, las vistas apuntan a **CustomerList** como su ***DataContext*** y hacen uso del ***Data-Binding de WPF*** para enlazarse a la lista de Clientes.

También se puede ver que hemos sustituido `List<Customer>` por `ObservableCollection<Customer>` para permitir a las vistas enlazarse con **CustomerList** haciendo uso de mecanismos de **data-binding.**

Nota:
Destacar que el mecanismo de data-*binding* de WPF nos permite crear un diseño mucho más desacoplado (por lo menos hasta que tengamos que añadir cierta lógica de negocio cliente)

Problemas del Diseño Simplista WPF

El problema es que las cosas se complican cuando introducimos la siguiente funcionalidad:

1.- Propagación del elemento seleccionado, de forma que actualizamos la vista **DetailsView** siempre que la selección de un elemento en **ListView** cambia, y viceversa.

2.- Habilitar o deshabilitar partes de Interfaz de Usuario de **DetailsView** o **ListView** basado en alguna regla (por ejemplo, resaltar una entrada que tiene un ZIP no perteneciente a España).

Las característica 1 puede implementarse añadiendo una propiedad `CurrentEntry` directamente en `CustomerList`. Sin embargo, esta solución ofrece problemas. Si tenemos más de una instancia del UI conectado al mismo **CustomerList**.

La característica 2 puede implementarse en las vistas (ListView y DetailsView), pero el problema si hacemos eso es que si queremos cambiar la regla, entonces necesitaremos cambiar ambas vistas. Los cambios nos empezarán a impactar en múltiples sitios.

En definitiva, de forma gradual parece conveniente que necesitamos una tercera sub-capa en nuestra aplicación. Necesitamos una sub-capa entre las vistas y el modelo CustomerList que guarda los estados compartidos entre las vistas. Necesitamos el concepto de 'Modelo de Vistas' ó **ViewModel**.

5.2.2.-Diseño con patrón Model-View-ViewModel (MVVM)

Un ViewModel nos sirve de la tercera capa intermedia que necesitamos, nos proporciona una abstracción que actúa como una meta-vista (un modelo de una vista), guardando estados y políticas que son compartidas por una o un conjunto de Vistas.

Al introducir el concepto de **ViewModel**, nuestro diseño quedaría como el siguiente:

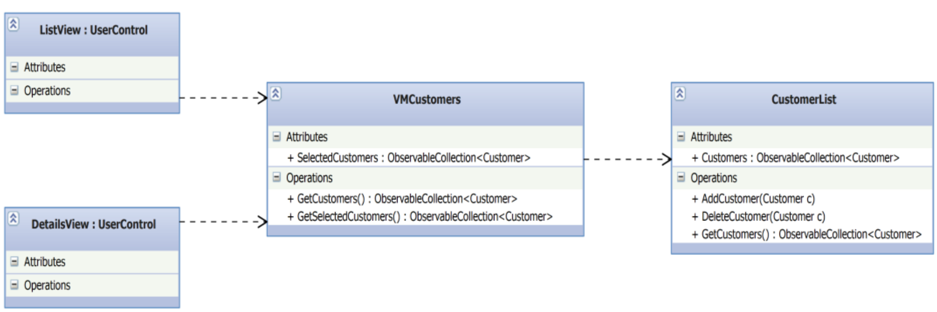


Figura 18.- Diseño de aplicación usando un VIEW-MODEL

En este diseño, las vistas conocen el **ViewModel** y se enlazan a sus datos, para poder reflejar cualquier cambio que tenga. <u>El **ViewModel** no tiene ninguna referencia a las **Vistas**, solo tiene una referencia al **Modelo,**</u> en nuestro caso **CustomerList**.

Para las Vistas, el ViewModel actúa como fachada del modelo pero también como una forma de compartir estados entre vistas (`selectedCustomers` en el ejemplo). Adicionalmente, el ViewModel expone normalmente 'Commands' a los que las Vistas pueden enlazarse.

Uso de 'Commands' en aplicaciones MVVM WPF

WPF implementa el patrón de diseño Commands como mecanismo de programación de entrada de eventos. Los comandos permiten desacoplamiento entre el origen y la acción gestionada con la ventaja de que múltiples fuentes (controles de vistas) pueden invocar al mismo comando, y el mismo comando puede ser gestionado también de forma diferente dependiendo del objetivo.

Desde el punto de vista del uso, un comando es sólo una propiedad de un elemento de la interfaz de usuario que lanza un comando lógico en vez de invocar un manejador de eventos directamente (recuerde el problema expuesto en la presentación de este apartado). Podemos tener múltiples componentes de esa interfaz de usuario lanzando el mismo comando. Pero además, podemos hacerlo directamente desde el XAML, por lo que un diseñador sólo ve una orden y un programador tendrá en alguna parte de su código (que no es recomendable que esté en el codebehind) la funcionalidad del comando.

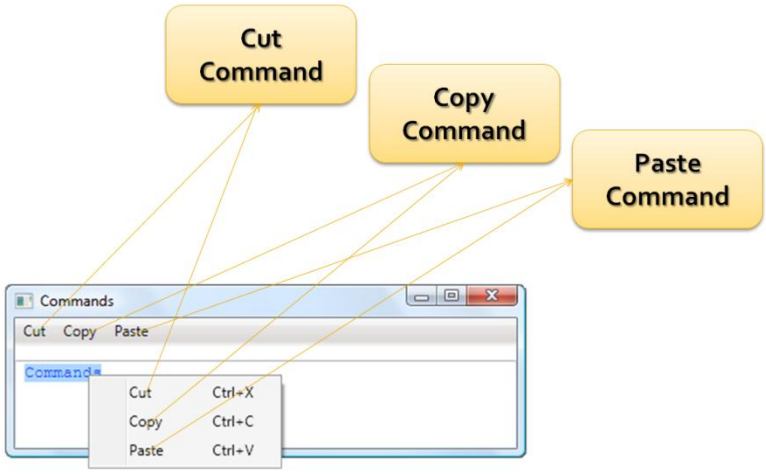


Figura 19.- Inserte el texto que desee

Existen además más funcionalidades que se le podrían pedir a los Comandos como por ejemplo, que si una acción no está disponible no se puede lanzar. Por ejemplo, si no hemos modificado nada de un documento ¿Para qué vamos a activar la acción Deshacer? Los componentes XAML de nuestra aplicación que lanzan la acción Deshacer deberían de estar desactivados.

Todos los 'commands' implementan en interfaz **ICommand**.

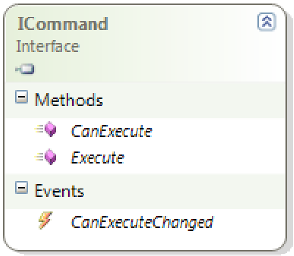


Figura 20.- Interfaz ICommand

Este interfaz está constituido por dos métodos y un evento.

- **Execute**: Que contiene la lógica de la acción que debe implementar el comando en la aplicación.
- **CanExecute**: Sirve para devolver el estado del comando, pudiendo comunicar si este está habilitado o no.
- **CanExecuteChanged**: Siempre que el valor de CanExecute cambie se lanzará un evento informando de ello.

Un ejemplo de implementación de un comando sería:

```
public class SaveCommand : ICommand
    {
        private CustomerViewModel _view;

        public SaveCommand(CustomerViewModel view)
        {
            _view = view;
        }

        public bool CanExecute(object parameter)
        {
            return true;
        }

        public event EventHandler CanExecuteChanged;

        public void Execute(object parameter)
        {
            //Hacer algo, como guardar un cliente
        }
    }
```

No obstante, para acciones de Interfaz Gráfico, WPF proporciona una clase especial llamada **RoutedCommand** que un objeto comando que no sabe cómo realizar la tarea que representa.

Cuando se le pregunta si se puede ejecutar (CanExecute) o cuando se ordena que se ejecute (Execute), el routedCommand delega (Delegate) la responsabilidad a otro. Los comandos enrutados viajan a través del árbol visual de elementos de WPF dando la oportunidad a cada elemento de la UI de que ejecute el comando que realiza el trabajo. Además, todos los controles que usan RoutedCommand pueden estar automáticamente “disable” cuando no se puedan ejecutar.

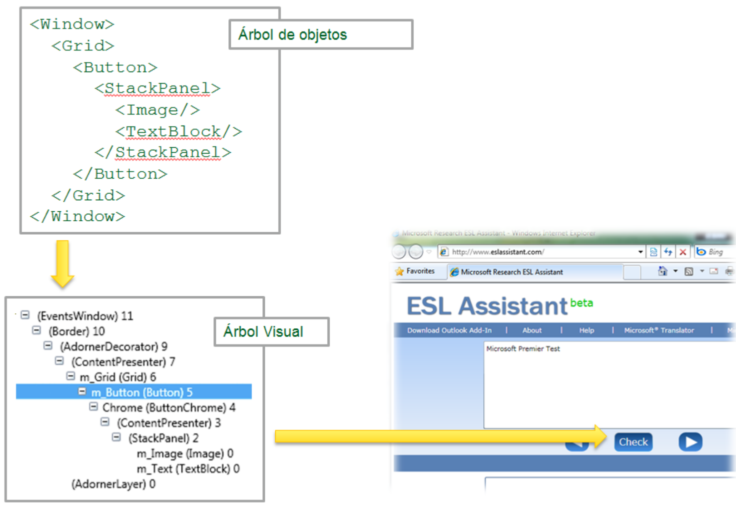


Figura 21.- RoutedCommand en WPF

En diseños MVVM, sin embargo, los objetivos de commands se implementan a menudo en los ViewModels, que sin embargo nos son típicamente parte del árbol de elementos visuales. Esto requiere la introducción de un comando de diferente clase: el comando enlazable (**DelegateCommand)** que implementa **ICommand** y puede tener elementos no visuales como sus objetivos.

Teniendo en cuenta los comandos enlazables, el diseño aplicando MVVM quedaría así:



Figura 22.- Diseño de aplicación usando un VIEW-MODEL y exponiendo COMMANDS enlazables-

Un **DelegateCommand** permite a las vistas enlazarse al **ViewModel**. Aunque el diagrama anterior no lo muestra explícitamente, todos los **ICommand** expuestos por el

ViewModel son **DelegateCommands**. Por ejemplo, el código para el **AddCustomerCommand** en el **ViewModel** sería algo así:

C#

```
public class VMCustomerList : ObservableObject
{
```

Propiedad **ICommand** para Edición del Customer seleccionado

```
   private ICommand _editCommand;

   public ICommand EditCommand
   {
      if (_editCommand == null)
      {
```

Creación del **DelegateCommand**

```
         _editCommand   =   new   DelegateCommand<Customer>(EditExecute,
   CanEditExecute);
      }
        return _editCommand;
   }

      private void EditExecute(...) { ... }
      private bool CanEditExecute() { ... }
   }
```

Y el código XAML que hace uso de este COMMAND, simplificándolo, sería algo así:

XAML de View

Binding con **Command** EditCommand

```
<UserControl>
...
<StackPanel>
...
<Button  Content="Button"  Command="{Binding  EditCommand,  Mode=OneWay}"
CommandParameter="{Binding SelectedItem, ElementName=listBox}"/>
...
</StackPanel>
...
</UserControl>
```

Uso de INotifyPropertyChanged en aplicaciones MVVM WPF

Por último, recordar que en WPF existe una interfaz llamada INotifyPropertyChanged, que se puede implementar para notificar a la interfaz de usuario que las propiedades de un objeto se han modificado, y que por lo tanto la interfaz debe actualizar sus datos. Todo este mecanismo de suscripción lo hacen los enlaces a datos de WPF de forma automática. Cuando queremos devolver una colección de objetos usamos, como explicamos anteriormente, la colección observable (ObservableCollection). Pero cuando tenemos que pasar del modelo, a la vista, pasando por el ViewModel un solo objeto, tendremos que hacer uso de esta Interfáz.

Esta interfaz define un solo evento, llamado PropertyChanged que debe lanzarse para informar del cambio de propiedad. Es responsabilidad de cada clase del modelo lanzar el evento cuando sea oportuno:

```
public class A : INotifyPropertyChanged
{
    private string _name;

    // Evento definido por la interfaz
    public event PropertyChangedEventHandler PropertyChanged;

    // Lanza el evento "PropertyChanged"
    private void NotifyPropertyChanged(string info)
    {
        var handler = this.PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(info));
        }
    }
    // Propiedad que informa de sus cambios
    public string Name
    {
        get { return _name; }
        set
        {
            if (_name != value)
            {
                _name = value;
                NotifyPropertyChanged("Name");
            }
        }
    }
}
```

Este código es pesado de realizar en clases con muchas propiedades y es fácil cometer errores. Por lo que es importante crearse una pequeña clase que nos evite tener que repetir el mismo código una y otra vez, y por tanto se recomienda realizar algo como esto:

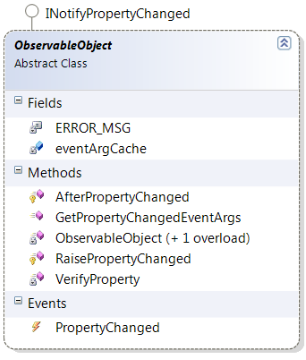


Figura 23.- Interfaz INotifyPropertyChanged

Uso de INotifyPropertyChanged en aplicaciones MVVM WPF

Una de las tareas más comunes que se va a realizar en aplicaciones MVVM es que desde el modelo de la vista (ViewModel) vamos a gestionar la carga de distintas vistas dependiendo del tipo de comando que se haya ejecutado. Si por ejemplo pulsamos un botón en una vista cuyo comando es MostrarDetalles, casi seguro que necesitaremos irnos desde ese ViewModel que opera el comando a otra vista (MostrarDetallesView) que a su vez tendrá otro ViewModel (VMMostrarDetalles).

Siempre vamos que tener que realizar las mismas operaciones de Navegar a otra vista y asignar el nuevo ViewModel, por lo tanto se recomienda realizar una clase que implemente dicha funcionalidad.

```
        public static void NavigateToView(UserControl view)
         {
                     NavigateToView(view, null);
         }

         public static void NavigateToView(UserControl view, ObservableObject
    viewModel)
         {
              if (view != null)
              {

    ((MainWindow)App.Current.MainWindow).ContentHolder.Children.Remove(Navig
    ationController.currentView);

                  if ( viewModel != null )
                            view.DataContext = viewModel;


    ((MainWindow)App.Current.MainWindow).ContentHolder.Children.Add(view);
                         NavigationController.currentView = view;
              }
         }
```

5.3.- Implementación del patrón MVVM en Silverlight 4.0

Existen dos diferencias claras a la hora de implementar el patrón MVVM en Silverlight con respecto a las pautas que ya hemos visto en aplicaciones WPF.

Estas dos diferencias se pueden dividir en estas categorías:

1. Modelo de programación asíncrona
2. Modelo de validaciones

5.3.1.- Modelo de programación asíncrona

Una de las ventajas principales del uso de la arquitectura de interfaz de usuario creada a partir del patrón Model-View-View Model es la de aprovechar su modelo de asincronía para crear interfaces que respondan incluso al realizar tareas pesadas. Evitar tiempos de demora entre petición y respuesta por sobrecargas en la red o tareas que requieren de cálculos de gran complejidad es uno de los grandes beneficios del uso de un **ViewModel**.

- Un objeto **ViewModel** se ejecuta en un hilo diferente del encargado de construir la interfaz (**Dispatcher Thread**), por lo que cualquier acción iniciada a través de un objeto **ViewModel** no interferirá en el proceso de dibujado de la pantalla ni tampoco en las transiciones visuales que ésta lleva a cabo.

- Silverlight a su vez nos obliga a usar un modelo de proxy asíncronos para consumir los servicios referenciados por nuestra aplicación haciendo uso de la interfaz **IAsyncResult**, por lo que se diferencia del modelo de consumo de servicios usado por las aplicaciones WPF.

Estas dos características proporcionan un modelo de aplicación que nos lleva inequívocamente a la consecución de una aplicación con un alto grado de respuesta de las interacciones del usuario final.

Veamos un ejemplo de cómo se crean (de generación automática) los métodos de consumo del objeto proxy cliente de nuestra aplicación Silverlight:

```
private void OnGetPagedBankAccountsCompleted(object state) {
               if ((this.GetPagedBankAccountsCompleted != null)) {
                   InvokeAsyncCompletedEventArgs e =
((InvokeAsyncCompletedEventArgs)(state));
                   this.GetPagedBankAccountsCompleted(this, new
GetPagedBankAccountsCompletedEventArgs(e.Results, e.Error, e.Cancelled,
e.UserState));
               }
           }

           Public void
GetPagedBankAccountsAsync(Microsoft.Samples.NLayerApp.Presentation.Silve
rlight.Client.ServiceAgent.PagedCriteria pagedCriteria) {
               this.GetPagedBankAccountsAsync(pagedCriteria, null);
```

Del mismo modo que encontramos diferencias en el objeto proxy que generamos para consumir el servicio que expone los datos de nuestro modelo, existen diferencias en el modo de implementar las clases **ViewModel**.

Mientras la implementación base de nuestro **ViewModel** permanece inalterado con respecto a *Windows Presentation Foundation*, la manera de consumir los servicios si contempla un cambio en el cuerpo de la acción **Execute**() de los **RoutedCommand**.

En *Windows Presentation Foundation* hemos visto el uso del tipo **BackgroundWorker** para impedir que el hilo principal de nuestra aplicación se bloquee impidiendo la interacción con nuestra interfaz, el modelo de consumo de servicios de Silverlight ya nos expone de manera natural dicha asincronía.

Veamos dicha diferencia comparando la implementación de la acción **Execute**() en las dos tecnologías:

Ejecución de la acción *GetPagedCustomer* en WPF:

```
   using (BackgroundWorker worker = new BackgroundWorker())
               {
                   worker.DoWork += delegate(object sender,
DoWorkEventArgs e)
                   {
                       MainModuleServiceClient client = new
MainModuleServiceClient();

                       e.Result = client.GetPagedCustomer(new
PagedCriteria() { PageIndex = this._pageIndex, PageCount = 10 });
                   };

                   worker.RunWorkerCompleted += delegate(object sender,
RunWorkerCompletedEventArgs e)
                   {
                       if (!e.Cancelled && e.Error == null)
                       {
                           List<Customer> customers = e.Result as
List<Customer>;

                           if (customers != null)
                           {
                               this.Customers = new
ObservableCollection<Customer>(customers);
                               this._viewData =
CollectionViewSource.GetDefaultView(this.Customers);
                               this._viewData.Filter = null;
                           }
                       }
                       else
                           MessageBox.Show(e.Error.Message, "Customer
List", MessageBoxButton.OK, MessageBoxImage.Error);
                   };

                   worker.WorkerSupportsCancellation = true;
                   worker.RunWorkerAsync();
               }
```

Ejecución de la acción *GetPagedCustomer* en Silverlight:

```
   try
                {
                    MainModuleServiceClient client = new
MainModuleServiceClient();

                    client.GetPagedCustomerAsync(new PagedCriteria() {
PageIndex = 0, PageCount = 10 });

                    client.GetPagedCustomerCompleted += delegate(object
sender, GetPagedCustomerCompletedEventArgs e)
                    {
                        Customer[] listCustomers = e.Result;
                        if (listCustomers != null && listCustomers.Length
> 0)
                        {
                            Customer = listCustomers[0];
                            GetCustomerOrders();
                        }
                    };
                }
                catch (Exception excep)
                {
                    Debug.WriteLine("GetPagedCustomer: Error at Service:"
+ excep.ToString());
                }
```

5.3.2.- Modelo de validaciones

La segunda gran diferencia a la hora de abordar una implementación de una aplicación haciendo uso de Silverlight viene derivada de la implementación interna del modelo de validaciones.

WPF hace uso de tipos derivados de la clase abstracta **ValidationRule**, en las cuales definimos la lógica personalizada de validación de cliente y que vinculamos a un determinado enlace a datos (**Binding**). De este modo WPF hace uso de este objeto intermedio para validar los datos que viajan desde la Vista a nuestro **ViewModel** vinculado con dicha vista.

La principal desventaja del uso de los tipos **ValidationRule** deriva del comportamiento de la clase **Binding**. Aclaremos esto mejor; Hasta que el valor de la propiedad vinculada al objeto **Binding** no se modifica y este cambio es notificado a la interfaz de usuario, las reglas de validación no son comprobadas, por lo que si queremos hacer uso de ellas de un modo más explícito, deberemos recorrer el árbol de objetos para invocar dichas validaciones de manera procedural.

En Silverlight este modelo de validaciones se ha modificado para dar un mayor control sobre dicha tarea a través de la implementación en nuestros **ViewModel** de la interfaz **INotifyDataErrorInfo**.

Entraremos más en detalle sobre el uso de esta interfaz en el apartado de validaciones.

5.4.- Beneficios y Consecuencias del uso de MVVM

El uso del patrón MVVM proporciona varios **beneficios** fundamentales:

- Un ViewModel proporciona un único almacén de estados y políticas de presentación, lo que mejora la reutilización del Modelo (desacoplándolo de las Vistas) y facilita el reemplazamiento de las Vistas (al eliminar políticas específicas de presentación, de las Vistas)

- Un diseño MVVM mejora la facilidad de realizar *Testing* (Pruebas Unitarias especialmente) de la aplicación. Al separar la lógica de las vistas y controles visuales, podemos crear fácilmente pruebas unitarias que se encarguen exclusivamente del Modelo y del *ViewModel* (Puesto que las Vistas serán normalmente solo XAML, sin *'code-behind'*). Adicionalmente, MVVM también facilita la implementación de MOCKs en la Capa de Presentación, porque habiendo desacoplado las Vistas del Modelo y situando la lógica cliente en los ViewModels, esa lógica es fácilmente sustituible por MOCKs (simulación de ejecución), que es fundamental para el *testing* de aplicaciones complejas.

- El patrón MVVM ofrece un diseño desacoplado. Las Vistas solo tienen referencia al ViewModel y el ViewModel referencia solo al Modelo. El resto lo realiza el *databinding* y los *Commands* de la infraestructura de WPF.

Las **consecuencias** provocadas por el uso del patrón MVVM son:

- La relación típica entre un ***ViewModel*** y sus correspondientes ***Vistas*** son normalmente 'una a muchas', pero hay situaciones en las que eso no es cierto. En general, cualquier lógica de negocio cliente y lógica de negocio de validación de entrada de datos (seguimiento de selección de elementos, etc.) debe implementarse en el *ViewModel*.

- Hay situaciones donde un ***ViewModel*** es 'consciente' de otro ViewModel dentro de la misma aplicación. Esas situaciones aparecen cuando hay una relación maestro-detalle entre dos *ViewModels* o cuando un *ViewModel* representa a un elemento suelto (por ejemplo, la representación visual de un único Cliente). Cuando esto sucede, un *ViewModel* puede representar a una colección de *ViewModels*, como el caso demostrado anteriormente.

6.- VALIDACIÓN DE DATOS EN LA INTERFAZ (WPF)

¿Qué son las reglas de Validación?

Cuando queremos comprobar que el valor introducido por el usuario en la interfaz es el correcto o el que se espera, en WPF disponemos de las denominadas Reglas de Validación.

Mediante el uso de Reglas de Validación podemos validar el valor introducido por el usuario antes de que se actualice la propiedad enlazada. Por defecto, disponemos de la regla *ExceptionValidationRule* que nos permite capturar excepciones cuando se actualiza la fuente de datos del binding. El motor de bindings de WPF comprobará cada binding con su correspondiente regla de validación cada vez que se cambie un valor, es decir, cualquiera de las propiedades de la vista modelo en nuestro caso, y que dependerá de la configuración del propio binding, es decir, de qué valor tengan las propiedades *Mode* y *UpdateSourceTrigger*.

También podemos crear nuestras propias reglas de validación implementando la clase abstracta *ValidationRule*, cuyo método Validate será el que ejecutará el motor de bindings para comprobar si el valor es correcto o no. Es más, en el resultado de respuesta podemos incluir un mensaje de error que podremos mostrar en la interfaz al usuario para darle más información de porqué el dato introducido no es válido.

```
public class RequiredValidationRule: ValidationRule
{
        public override ValidationResult Validate(object value, System.G
lobalization.CultureInfo cultureInfo)
        {
            if (value == null )
                return new ValidationResult(false,
                                             "Este campo es requerido");

            if (string.IsNullOrEmpty(value.ToString().Trim()))
                return new ValidationResult(false,
                                             "Este campo es requerido");

            return new ValidationResult(true, null);
        }
    }
```

La manera de usar una regla de validación es muy sencilla:

- Lo primero que deberemos hacer es añadir el xml namespace de las reglas de validación personalizadas en la vista en que vayamos a utilizarlas. En el ejemplo, corresponde a la vista AddCustomerView.

```
xmlns:ValidationRules="clr-
namespace:Microsoft.Samples.NLayerApp.Presentation.Windows.WPF.Client.Va
lidationRules"
```

- Una vez añadido buscamos el binding donde queremos incluir la regla de validación y lo modificamos de la siguiente manera.

```
<TextBox Margin="0" TextWrapping="Wrap" Grid.Column="1" Style="{DynamicR
esource TextBoxStyle1}" x:Name="txtCompanyName" GotFocus="txtCompanyName
_GotFocus" LostFocus="txtCompanyName_LostFocus">
<Binding Path="CompanyName" Mode="TwoWay" NotifyOnValidationError="True"
 UpdateSourceTrigger="PropertyChanged" ValidatesOnDataErrors="True" Vali
datesOnExceptions="True">
           <Binding.ValidationRules>
               <ValidationRules:RequiredValidationRule />
          </Binding.ValidationRules>
     </Binding>
</TextBox>
```

- A continuación debemos editar el estilo del control TextBox estableciendo la propiedad *Validation.ErrorTemplate* para que, en nuestro caso, aparezca con un borde rojo la caja de texto con un error. Además vamos a añadir un trigger para que cuando la propiedad *Validation.HasError* sea verdadera, se establezca el tooltip de la caja de texto con el mensaje de error.

```
<Style BasedOn="{x:Null}" TargetType="{x:Type TextBox}">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="{x:Type TextBox}">
                (...)
            </ControlTemplate>
        </Setter.Value>
    </Setter>
    <Setter Property="Validation.ErrorTemplate">
        <Setter.Value>
            <ControlTemplate>
                <Border      BorderBrush="Red"      BorderThickness="1"
CornerRadius="5">

<AdornedElementPlaceholder></AdornedElementPlaceholder>
                </Border>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
    <Style.Triggers>
        <Trigger Property="Validation.HasError" Value="true">
            <Setter Property="ToolTip"
            Value="{Binding RelativeSource={RelativeSource Self},
            Path=.,
            Converter={StaticResource errorInfo}}"/>
        </Trigger>
    </Style.Triggers>
</Style>
```

Ahora que tenemos la parte visual preparada, nos queda modificar el código de la vista para que su viewmodel correspondiente se "entere" de que hay o no errores y actuar en consecuencia. Para esto disponemos de un evento al que nos podemos suscribir. El evento en cuestión es **Validation.ErrorEvent** que se lanzará cuando haya un error de validación pero solo en aquellos bindings donde la propiedad **NotifyOnValidationError** sea verdadero.

Seguimos en la vista AddCustomerView y nos suscribimos al evento antes mencionado.

```
            private Dictionary<string, bool> _errors = new
Dictionary<string, bool>();

            private void AddCustomerView_Loaded(object sender,
RoutedEventArgs e)
            {
                //add handler for validation errors
                this.AddHandler(Validation.ErrorEvent, new
RoutedEventHandler(OnErrorEvent));
            }

            private void OnErrorEvent(object o, RoutedEventArgs args)
            {
                if (args.OriginalSource != null)
                {
                    TextBox txtBox = (TextBox)args.OriginalSource;

                    if (!_errors.Keys.Contains(txtBox.Name))
                        _errors.Add(txtBox.Name, false);

                    _errors[txtBox.Name] =
Validation.GetHasError(txtBox);
                }

                this.IsValidData = (this._errors.Where(k => k.Value ==
true).Count() == 0);
            }
```

Lo que estamos haciendo cuando se lanza el evento es comprobar si el elemento tiene error y lo vamos registrando en un diccionario de elemento para controlar los que han modificado su estado de error y los que no. Finalmente comprobamos si hay elementos con error y asignamos el valor a la propiedad IsValidData.

Nota:

Podemos forzar la validación de una caja de texto mediante el uso del método UpdateSource():

this.TB_ZIPCode.GetBindingExpression(TextBox.TextProperty).UpdateSource()

Nota:

Es probable que si implementas el mismo sistema para avisar visualmente al usuario que hay un error en una caja de texto, con un borde rojo, utilizando AdornedElementPlaceholder cuando muestres elementos que se superponen a la caja de texto con error, el borde se siga viendo. Esto es debido a que el AdornedElementPlaceholder buscará en el árbol visual el primer AdornerDecorator que haya para pintarse. Al no especificar ninguno, todo el árbol comparte AdornerDecorator por lo que el borde se superpone a todo lo que pongas por encima de la caja de texto. Para evitar esto lo más sencillo es colocar un AdornerDecorator en el control de usuario que contiene las cajas de texto donde se hacen las validaciones, por ejemplo.

7.- VALIDACIÓN DE DATOS EN LA INTERFAZ DE USUARIO (SILVERLIGHT)

La validación de datos de interfaz de usuario en Silverlight 4 utilizando MVVM difiere con lo que acabamos de ver en WPF.

En Silverlight 4 disponemos de una interfaz **INotifyDataErrorInfo** que utilizaremos en nuestras Vistas Modelo. La ventaja con respecto a WPF es que nos podemos despreocupar de la parte visual ya que Silverlight 4 dispone de un sistema de para mostrar al usuario que existe un error.

Para utilizar **INotifyDataErrorInfo** simplemente debemos implementar dicha interfaz. Nosotros la hemos implementado en la clase **ObservableObject** que es la base de todas os viewmodels. La implementación es muy sencilla. Lo que hemos hecho ha sido crear una serie de métodos de ayuda para añadir a un diccionario las propiedades con error y mantener un control al igual que en el caso de WPF.

```
    public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;
            private     readonly     Dictionary<string,     List<string>>
_currentErrors;

            public IEnumerable GetErrors(string propertyName)
            {
                if (string.IsNullOrEmpty(propertyName))
                    return (_currentErrors.Values);

                MakeOrCreatePropertyErrorList(propertyName);
                return _currentErrors[propertyName];
            }

            public bool HasErrors
            {
                get
                {
                    return (_currentErrors.Where(c => c.Value.Count >
0).Count() > 0);
                }
```

```
            }

            void FireErrorsChanged(string property)
            {
                if (ErrorsChanged != null)
                    ErrorsChanged(this, new
DataErrorsChangedEventArgs(property));
            }

            public void ClearErrorFromProperty(string property)
            {
                MakeOrCreatePropertyErrorList(property);
                _currentErrors[property].Clear();
                FireErrorsChanged(property);
            }

            public void AddErrorForProperty(string property, string
error)
            {
                MakeOrCreatePropertyErrorList(property);
                _currentErrors[property].Add(error);
                FireErrorsChanged(property);
            }

            void MakeOrCreatePropertyErrorList(string propertyName)
            {
                if (!_currentErrors.ContainsKey(propertyName))
                    _currentErrors[propertyName] = new List<string>();
            }
```

En próximas versiones de los ejemplos se crearán manejadores de reglas de validación, pero en esta versión solo dispondremos de una regla de requeridos implementada en la misma clase **ObservableObject**.

La regla es muy sencilla, comprueba que la propiedad tenga valor y en caso contrario añade dicha propiedad al diccionario de errores y lanza el error.

```
           public void CheckRequiredValidationRule(string propertyName,
string value)
            {
                ClearErrorFromProperty(propertyName);
                if (string.IsNullOrEmpty(value) ||
string.IsNullOrWhiteSpace(propertyName))
                    AddErrorForProperty(propertyName, "Este campo es
requerido");
                RaisePropertyChanged(propertyName);
    }
```

Finalmente lo único que nos queda es lanzar la regla de validación en la propiedad que queramos validar de la siguiente manera:

```
    public string CompanyName
    {
        get { return _currentCustomer.CompanyName; }
        set
        {
            _currentCustomer.CompanyName = value;
```

```
                _currentCustomer.CustomerCode = (value.ToString().Length > 3)
    ? value.ToString().Substring(0, 3) : null;
                CheckRequiredValidationRule("CompanyName", value);
            }
        }
```

Y realizar el binding en la propiedad **Text** de la caja de texto pero sin olvidar establecer a verdadero la propiedad del binding **ValidatesOnNotifyDataErrors**.

```
        {Binding CompanyName, Mode=TwoWay, ValidatesOnNotifyDataErrors=True}
```

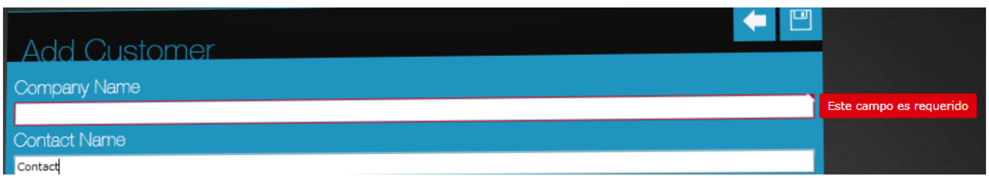


Figura 24.- Validación en Silverlight

8.- IMPLEMENTACIÓN CON ASP.NET MVC 2.0

ASP.NET MVC es la implementación del patrón modelo vista controlador para la web. Esta implementación se basa en la plataforma ASP.NET y por tanto permite usar muchos de los componentes existentes como la autenticación, la autorización o el caché.

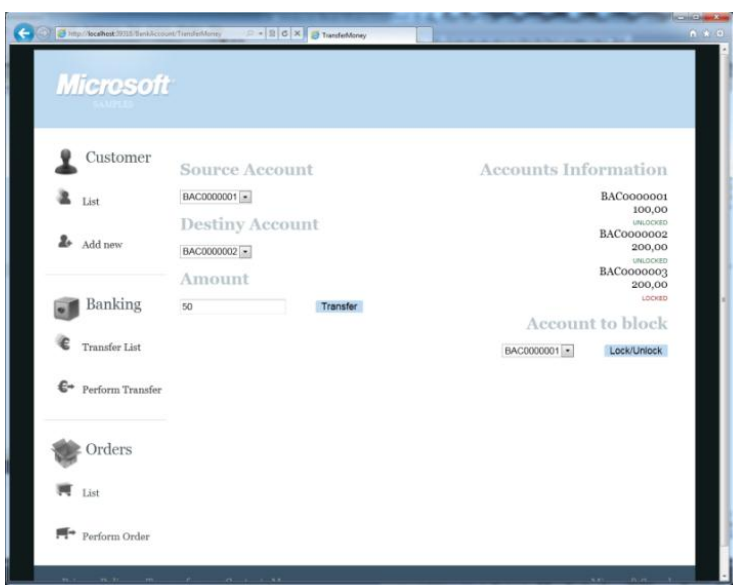


Figura 25.- MVC – 'Transferencias Bancarias'

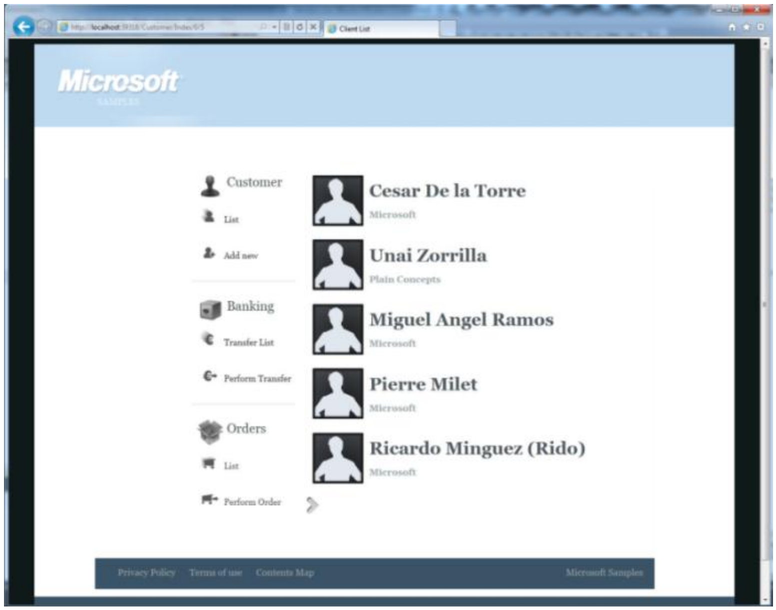


Figura 26.- MVC – Vista de 'lista de clientes'

8.1.- Fundamentos de ASP.NET MVC

ASP.NET MVC organiza la interacción del usuario con el sistema en controladores que exponen acciones que se encargan de modificar o consultar el modelo del sistema. Estas acciones devuelven un resultado que tras ejecutarse devuelve una respuesta al usuario. Veamos más en detalle el pipeline para entenderlo más fácilmente.

8.2.- El pipeline de ASP.NET MVC

Cuando llega una petición HTTP, el sistema de Routing intenta encontrar un controlador y acción asociados para responder a la petición en base al contenido y la url de la misma. Una vez encontrado el controlador adecuado, pide una instancia del mismo a la fábrica de controladores. Una vez dispone de una instancia del controlador, busca dentro del mismo la acción que debe invocar, mapea los parámetros de la acción en base al contenido de la petición (modelbinding) e invoca la acción con los parámetros obtenidos. La acción del controlador interactúa con el sistema y devuelve un resultado. Este resultado puede ser un fichero, una redirección a otro controlador,

una vista, etc. El caso más común es que sea una vista (ViewResult). Los resultados de una acción son derivados de ActionResult, y es importante entender que simplemente describen el resultado de la acción y que no ocurre nada hasta que son ejecutados. En el caso de un ViewResult, al ejecutarse, se invoca a un ViewEngine que se encarga de encontrar la vista solicitada y renderizar el HTML que se envía por respuesta.

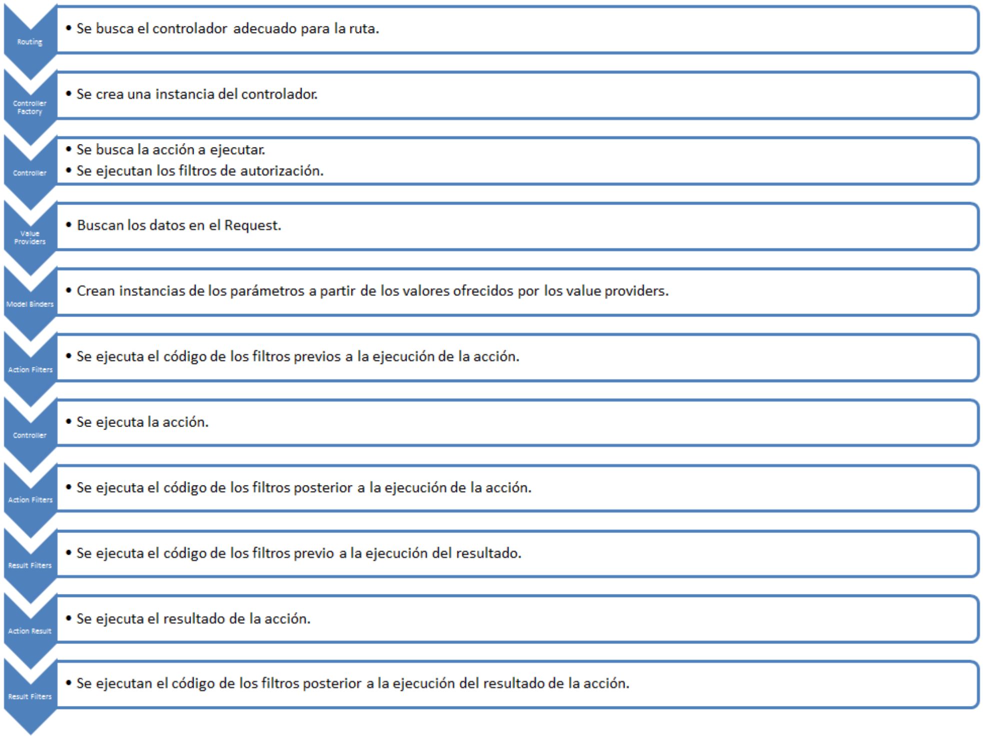


Figura 27.-

Un controlador se comunica con una vista a través de un objeto llamado ViewData que es un diccionario encargado de almacenar toda la información necesaria para renderizar la vista. Las vistas pueden ser fuerte o débilmente tipadas. En caso de ser fuertemente tipadas la vista es una clase genérica cerrada (todos los parámetros de tipo han sido especificados) y el ViewData expone una propiedad Model con el tipo del parámetro genérico de la vista.

De esta forma se expresa mejor la intención de la vista, queda más claro y explícito su contrato con cualquier controlador y se dispone de Intellisense al acceder a los miembros del modelo mientras creamos la vista.

8.3.- Un ejemplo completo: Actualización de un cliente

Lo primero que haremos será delinear el escenario que vamos a implementar. Típicamente en cualquier aplicación MVC actualizar una entidad requiere dos acciones, una para visualizar un formulario con la entidad que queremos modificar y otra para enviar los cambios al servidor. Es importante destacar que en este tipo de aplicaciones intentamos seguir un protocolo REST (REpresentational State Transfer) y esto se verá reflejado en los controladores. Veamos primero como son las dos acciones:

```
public ActionResult Edit(string customerCode)
{

Customer customer = _customerService.FindCustomerByCode(customerCode);
    return View(customer);
   }

   [HttpPost]
public ActionResult Edit(Customer customer)
{
    try
    {
        _customerService.ChangeCustomer(customer);
        return RedirectToAction("Details", new

{ customerCode = customer.CustomerCode });

    }
    catch
    {
            return View();
    }
}
```

Como podemos ver tenemos una primera acción para solicitar el cliente que queremos editar y una segunda acción para enviar los cambios de vuelta. Todo el trabajo del controlador se divide en dos responsabilidades. Modificar o consultar el dominio, y decidir que pantalla se va a mostrar.

Veamos a continuación la vista de actualización. Si somos un poco perspicaces, nos daremos cuenta de que la primera acción se encarga de cargar el formulario con los datos actuales y la segunda acción se encarga de procesar los datos del formulario.

Si observamos la vista de edición de clientes, vemos que se divide en dos partes, por un lado la vista de edición correspondiente a la acción de editar, cuya responsabilidad es encajar la vista dentro del diseño global de la página y enlazar el formulario con su acción correspondiente:

```
<% using (Html.BeginForm("Edit","Customer",FormMethod.Post,
                              new { enctype = "multipart/form-data"}))
{ %>
   <fieldset>
   <legend><%: ViewData.ModelMetadata.ModelType.Name %></legend>
    <%: Html.EditorForModel() %>
    <%: Html.SerializedHidden(Model) %>
    <input type="submit" value="Editar" />
    </fieldset>
<%} %>
```

Por otro lado la vista del formulario de cliente, cuya responsabilidad es mostrar los campos editables de un cliente y realizar y mostrar el resultado de la validación de los mismos:

```
   <div class="display-label">
       <%: Html.DisplayNameFor(x => x.CustomerCode) %></div>
   <div class="display-field">
       <%: Html.EditorFor(x => x.CustomerCode) %></div>
   <div class="validation-field">
       <%: Html.ValidationMessageFor(x => x.CustomerCode) %></div>
   <div class="display-label">
       <%: Html.DisplayNameFor(x => x.ContactName) %></div>
   <div class="display-field">
       <%: Html.EditorFor(x => x.ContactName) %></div>
   <div class="validation-field">
       <%: Html.ValidationMessageFor(x => x.ContactName) %></div>
   <div class="display-label">
       <%: Html.DisplayNameFor(x => x.ContactTitle) %></div>
   <div class="display-field">
       <%: Html.EditorFor(x => x.ContactTitle) %></div>
   <div class="validation-field">
       <%: Html.ValidationMessageFor(x => x.ContactTitle) %></div>
   <div class="display-label">
       <%: Html.DisplayNameFor(x => x.CompanyName) %></div>
   <div class="display-field">
       <%: Html.EditorFor(x => x.CompanyName) %></div>
   <div class="validation-field">
       <%: Html.ValidationMessageFor(x => x.CompanyName) %></div>
   <div class="display-label">
       <%: Html.DisplayNameFor(x => x.IsEnabled) %></div>
   <div class="display-field">
       <%: Html.EditorFor(x => x.IsEnabled,true) %></div>
   <div class="validation-field">
       <%: Html.ValidationMessageFor(x => x.IsEnabled) %></div>
   <div class="display-field">
       <%: Html.EditorFor(x => x.Country) %></div>
   <div class="display-field">
       <%: Html.EditorFor(x => x.Address) %></div>
   <div class="display-field">
       <%: Html.EditorFor(x => x.CustomerPicture) %></div>
```

El aspecto más importante a tener en cuenta en este tipo de aplicaciones es la gestión que hacemos del estado. Cuando actualizamos una entidad estamos realizando una modificación de un estado existente por lo que para aplicar adecuadamente los

cambios debemos disponer del estado original. Los dos posibles puntos de almacenamiento de este estado original son el servidor (base de datos o caché distribuida) o el cliente. Nosotros hemos optado por almacenar este estado original en el cliente, por lo que enviamos en la página una copia serializada de los datos originales, usando para ello un HtmlHelper que hemos diseñado a tal efecto y que podemos ver en la vista de edición:

```
<%: Html.SerializedHidden(Model) %>
```

Hemos terminado con la acción encargada de mostrar el formulario con los datos a editar, pero una vez hemos realizado cambios, ¿Cómo se aplican dichos cambios a la entidad? Si conocemos un poco el protocolo HTTP sabemos que cuando “Posteamos” un formulario, estamos enviando pares nombre=valor en la petición. Esto dista mucho de los parámetros de la acción encargada de salvar los cambios en la entidad que recibe directamente un objeto cliente. El proceso de traducción entre los elementos de una petición y los parámetros de una acción, es lo que se conoce como ModelBinding. ASP.NET MVC trae un modelbinder por defecto que se encarga de hacer esta traducción basándose en convenciones.

No obstante, dado que estamos empleando self-tracking entities, hemos creado nuestro propio ModelBinder para simplificar el proceso de actualización. Como es de esperar, un ModelBinder necesita crear una instancia de la clase a la que está mapeando los parámetros de la petición en algún momento. Gracias a la extensibilidad de ASP.NET MVC extendemos exactamente ese punto, y si hemos serializado la entidad original, devolvemos la misma en lugar de crear una copia nueva. Así al mapear los parámetros la entidad puede llevar la cuenta de qué parámetros han sido cambiados, simplificando el proceso de actualización:

```
    public class SelfTrackingEntityModelBinder<T> :
         DefaultModelBinder where T : class, IObjectWithChangeTracker
{
     protected override object CreateModel(ControllerContext controllerCo
ntext, ModelBindingContext bindingContext, Type modelType)
    {
        string steName = typeof(T).Name+"STE";
        if(bindingContext.ValueProvider.ContainsPrefix(steName)){
            var value = bindingContext.ValueProvider.GetValue(steName);
            return new SelfTrackingEntityBase64Converter<T>()
                              .ToEntity(value.AttemptedValue);
        }
        return base.CreateModel(controllerContext, bindingContext,
                                  modelType);
    }
   }
```

8.4.- Otros aspectos de la aplicación

Una aplicación MVC es un subtipo de aplicación ASP.NET, por lo que disponemos de un fichero global.asax que es el punto de entrada para nuestra aplicación. Típicamente en este fichero se suelen realizar todas las inicializaciones necesarias. En el caso de ASP.NET MVC estas inicializaciones suelen ser:

- Registro de rutas de mapeo URL / (Controlador, Acción).
- Registro de dependencias en el contenedor.
- Registro de factorías de controladores especializadas.
- Registro de motores de vistas alternativos.
- Registro de model binders específicos.
- Registro de value providers alternativos.
- Registro de áreas.

Una aplicación MVC puede dividirse en áreas para mejorar la organización de la misma. Cada área debería ser responsable de hacer todos los registros indicados que necesite. Este código en principio debe ir en el fichero global.asax, dentro del método Application_Start, pero como podemos presuponer, en una aplicación de gran tamaño este método puede alcanzar un número importante de líneas de código. Por eso, aunque no hemos creado áreas, hemos delegado la responsabilidad de cada "área" (consideraremos la aplicación como la única área definida) en un BootStrapper encargado de inicializar cada área. (Registrar dependencias en el contenedor, model binders, etc.)

Entrando en detalle, cada área es responsable de registrar sus dependencias en el contenedor de dependencias (principalmente los controladores). En las extensiones del framework está definida una fábrica que utiliza el contenedor de dependencias para crear los controladores e inyectar automáticamente sus dependencias.

CAPÍTULO

9

Capas de Infraestructura Transversal

1.- CAPAS DE INFRAESTRUCTURA TRANSVERSAL

La mayoría de aplicaciones contienen funcionalidad común que se utiliza en los las diferentes Capas tradicionales e incluso en diferentes Niveles físicos (*Tiers*). Este tipo de funcionalidad normalmente abarca operaciones como autenticación, autorización, cache, gestión de excepciones, *logging*/registros, trazas, instrumentalización y validación. **A este tipo de funcionalidad normalmente se le denomina 'Aspectos Transversales' o 'Aspectos Horizontales', porque afectan a la aplicación entera, y deben estar por lo tanto centralizados en una localización central si es posible, para favorecer la reutilización de componentes entre las diferentes capas**. Por ejemplo, el código que genera trazas en los ficheros de *log* de una aplicación, probablemente se utilice desde muchos puntos diferentes de la aplicación (desde diferentes capas y niveles físicos). Si centralizamos el código encargado de realizar las tareas específicas de generar trazas (u otra acción), seremos capaces en el futuro de cambiar el comportamiento de dicho aspecto cambiando el código solamente en un área concreta.

Este capítulo pretende ayudar a entender el rol que juegan en las aplicaciones estos aspectos transversales, identificar áreas que pueden identificarse en nuestras aplicaciones y aprender problemáticas concretas típicas al diseñar e implementar aspectos transversales.

Hay diferentes aproximaciones para diseñar e implementar estas funcionalidades, desde librerías de clases tipo 'bloques de construcción' a utilizar desde mis capas tradicionales, hasta AOP (*Aspect Oriented Programming*) donde se utilizan metadatos para bien insertar/inyectar código de aspectos transversales directamente en la compilación o bien durante el tiempo de ejecución (con intercepción de llamadas a objetos).

2.- SITUACIÓN DE INFRAESTRUCTURA TRANSVERSAL EN LA ARQUITECTURA

En el siguiente diagrama se muestra cómo encajan típicamente estos aspectos transversales en nuestra Arquitectura:

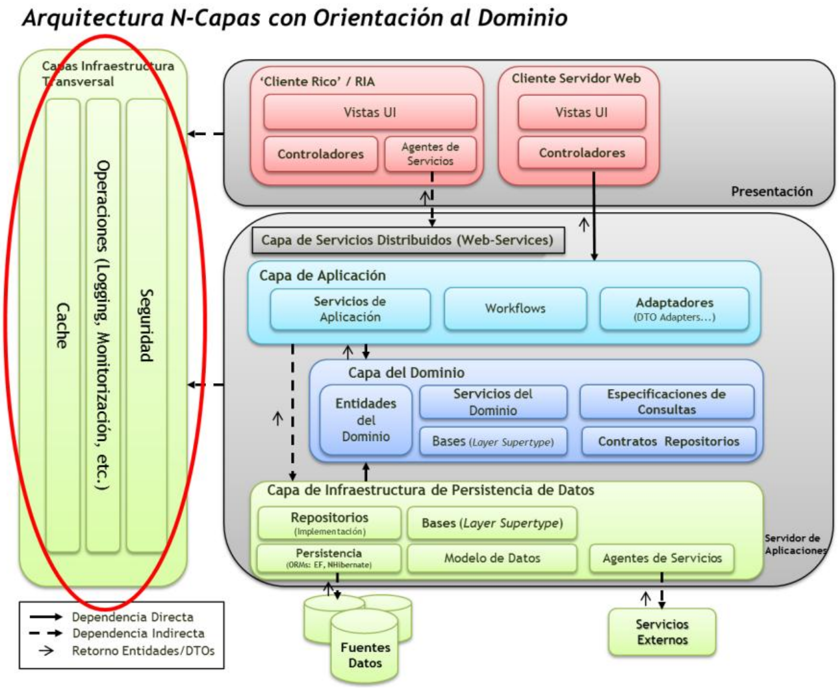


Figura 1.- Situación de Aspectos Transversales en Arquitectura N-Capas DDD

Siguiendo las directrices generales en DDD, lo denominamos definitivamente como "*Capas de Infraestructura Transversal*", porque estos componentes forman también parte de las Capas de Infraestructura o capas ligadas a una tecnología concreta. En definitiva, cada uno de dichos aspectos transversales trabaja con una tecnología concreta (Seguridad concreta, APIs específicos de instrumentalización, *logging*, etc.).

3.- CONSIDERACIONES GENERALES DE DISEÑO

Las siguientes guías pueden ayudar a entender los principales factores a considerar en esta área:

- Examinar las funciones requeridas por cada capa y buscar casos donde se pueda abstraer dicha funcionalidad a componentes comunes de propósito general para toda la aplicación que se puedan configurar dependiendo de requerimientos específicos de cada capa. Esos componentes pueden ser probablemente reutilizados en otra aplicación.

- Dependiendo de cómo se distribuyan físicamente los componentes y capas de la aplicación, podemos necesitar instalar componentes transversales en más de un nivel físico (*Tier*). Sin embargo, aun así nos beneficiamos de la reutilización y tiempos/coste de desarrollo menores.

- Considerar el uso de la técnica "*Inyección de Dependencias*" para inyectar instancias de componentes transversales, basándonos en información de configuración. Esto nos permitirá cambiar la ejecución concreta del aspecto transversal de una forma fácil y sin impacto al resto de la aplicación, incluso sin tener que recompilar ni tener que volver a desplegar la aplicación.

- Considerar el uso de librerías de terceras partes, normalmente altamente configurables, para la implementación de aspectos transversales comunes.

- Considerar AOP (*Aspect Oriented Programming*) para inyectar los aspectos transversales de una forma transparente y no entremezclada con la propia lógica de cada capa. Muchos de los contenedores IoC utilizados para 'Inyección de Dependencias', también ofrecen características de 'Intercepción de Llamadas' y por lo tanto, una forma de implementar 'aspectos AOP'.

Tabla 1.- Guía de Arquitectura Marco

Regla Nº: D29.	**Se identificarán áreas transversales de la aplicación e implementarán como Aspectos Horizontales/Transversales reutilizables por las diferentes capas.**

○ **Normas**

- Es importante identificar qué áreas de la aplicación serán aspectos transversales y así no realizar una práctica de copiar/pegar, sino una buena reutilización de componentes transversales.

Ventajas del uso de componentes transversales

→ Reutilización de Componentes

→ Homogenización de aspectos existentes en las diferentes capas de la aplicación con la consiguiente optimización del desarrollo

Referencias

'Crosscutting concerns' en '.Net Application Architecture Guide' de Microsoft Pattern & Practices:
http://www.codeplex.com/AppArch

4.- ASPECTOS TRANSVERSALES

Las siguientes áreas o aspectos son los más habituales a considerar como parte de estas capas de infraestructura transversal:

- Seguridad
    - Identidad
    - Autenticación
    - Autorización
    - Arquitectura de Seguridad orientada a *Claims*
- Cache
- Gestión de Configuración
- Gestión de Excepciones
- Instrumentalización
- *Logging*
- Gestión de Estados
- Validación de datos de entrada

4.1.- Seguridad en la aplicación: Autenticación y Autorización

Solamente teniendo en cuenta la seguridad de arquitectura y desarrollo de aplicaciones, podríamos cubrir un solo libro/guía de Arquitectura, pues existen innumerables conceptos a nivel de seguridad dependiendo del nivel (código seguro y agujeros de seguridad, comunicaciones seguras, cifrado, firma electrónica, etc.) En la presente guía no pretendemos cubrir todas las posibilidades en la seguridad de aplicaciones pues crearíamos un volumen adicional de documentación bastante exagerado sobre algo que no es el *Core* de esta Arquitectura N-Capas DDD.

Sin embargo, hay dos puntos fundamentales en la seguridad de una aplicación que siempre se deben cubrir, pues son necesarios para el uso de dicha aplicación por los usuarios. Se trata de la **Identidad** y **Autenticación** para identificar a los usuarios que acceden a la aplicación así como el concepto de **Autorización**, para comprobar y en su caso otorgar acceso a los usuarios sobre los recursos (áreas funcionales) de una aplicación.

Por último, relativo solo a la autenticación y autorización, a nivel de Arquitectura y tecnología, existen también bastantes posibilidades. Desde tipos de autenticación como el manido usuario-password, pasando por autenticación corporativa basada en *Windows Active Directory*, LDAPs, Certificados X.509, etc. así como sus relacionados métodos y tecnologías para implementar la autorización (Orientación a Roles de aplicación, donde tenemos diferentes opciones tecnológicas, *Roles .NET Windows, Roles .NET Custom, Roles-Membership, Authorization Manager, WIF*, etc.).

Debido a dicho gran volumen de variables, no pretendemos exponer todas las opciones arquitecturales de autenticación/autorización y ni mucho menos todas las opciones tecnológicas disponibles por tecnologías Microsoft para realizarlo. Por el contrario, hemos optado por seleccionar un tipo de Seguridad que sea el idóneo para el contexto al que nos dirigimos con esta guía, que es el **contexto de aplicaciones complejas, de envergadura** y en la mayoría de los casos por lo tanto, **aplicaciones que deben integrarse con la seguridad corporativa existente en una organización, la cual incluso puede ser variable**. Dicho tipo de seguridad seleccionada está basada en la '*Orientación a Claims*', en la que si entraremos en cierto detalle, tanto a nivel teórico/lógico como a nivel de implementación con tecnología Microsoft (WIF: *Windows Identity Foundation* y *ADFS 2.0*).

4.1.1.- Autenticación

El diseño de una correcta autenticación es fundamental para la seguridad de la aplicación. Si no se realiza correctamente, la aplicación podrá ser vulnerable a ataques

de 'spoofing', ataques de diccionario, secuestro de sesión y otros tipos de ataques. Considerar las siguientes guías relativas a la autenticación:

- Identificar fronteras de confianza y autenticar a los usuarios y llamadas que crucen dichas fronteras de confianza. Considerar qué puede requerirse autenticar tanto las llamadas originadas en cliente como las llamadas originadas en servidor (autenticación mutua).

- Si se hace uso de autenticaciones usuario/clave, las claves/passwords deben ser 'fuertes', es decir, que cumplan requisitos mínimos de complejidad (alfanuméricos, número mínimo de longitud, inclusión de números, etc.).

- Si se dispone de múltiples sistemas en la aplicación, o **si los usuarios deben acceder a múltiples aplicaciones con las mismas credenciales (requerimiento corporativo típico), considerar el uso de una estrategia de '*single sign-on*' relacionado con una 'orientación a claims'.**

- No transmitir nunca clases/*passwords* en texto plano por la red y no almacenar dichas claves en texto plano en una base de datos o almacén. En lugar de eso, guardar '*hashes'* de dichas *passwords*.

4.1.2.- Autorización

El diseño de una correcta autorización es fundamental para la seguridad de la aplicación. Incluso habiendo diseñado una correcta autenticación, si no se diseña e implementa correctamente el sistema de autorización, puede no servir de mucho la anterior Autenticación y dejar a la aplicación vulnerable a elevación de privilegios, descubrimiento de información y manipulación de datos no autorizada. Considerar las siguientes guías generales relativas a la autorización:

- Identificar las fronteras de confianza, usuarios autorizados y emisores de llamadas autorizados a pasar dichas fronteras de confianza.

- Proteger los recursos aplicando autorización a los llamadores basándonos en su identidad, grupos, roles e incluso idealmente *claims*. Minimizar el número de roles siempre que sea posible.

- 'Permisos & Roles' versus 'Roles': Si la autorización de la aplicación es compleja, considerar el uso de una granularización más fina que simplemente hacer uso de grupos/roles. Es decir, hacer uso de *permisos* requeridos para acceder a un recurso. Dichos permisos estarán a su vez asignados a roles de aplicación y paralelamente, los usuarios también estarán asignados a dichos roles de aplicación. El uso de permisos es algo

muy potente, pues deja desacoplados a los usuarios y roles de la implementación de la aplicación, pues en la aplicación los recursos solo estarán fuertemente acoplados a los *permisos* requeridos, no a los roles.

- Considerar el uso de autorización basada en recursos para realizar auditorías del sistema.

- Considerar el uso de **autorización basada en *Claims*** cuando se debe soportar una autorización federada en una mezcla de información como identidad, rol, derechos/permisos y otros factores. La autorización basada en Claims proporciona niveles adicionales de abstracción que simplifican el poder separar reglas de autorización de los propios mecanismos de autorización y autenticación. Por ejemplo, se puede autenticar a un usuario con un certificado o con un nombre-usuario y password y después pasar dicho conjunto de *claims* al servicio que determine el acceso a los recursos. La ventaja de la autorización basada en Claims es que dejamos desacoplada la autorización de nuestra aplicación del tipo de autenticación externa de los usuarios, pudiendo potencialmente aceptar cualquier tipo de tecnología de autenticación, gracias al papel intermediario de los STS (Security Token Service). **Estas últimas tendencias de autenticación: '*Orientación a Claims y STS*' son precisamente las opciones elegidas por la presente Guía de Arquitectura como sistema preferido** y se explicará en detalle su diseño e implementación.

4.1.3.- Arquitectura de Seguridad basada en 'Claims'

La gestión de identidades es desde todos los puntos de vista un verdadero reto. Existen miles de aplicaciones, tanto empresariales como sitios Web en Internet, y por lo tanto, existen también miles de tipos de credenciales. Pero el resumen, desde el punto de vista de un usuario podría ser: "No quiero estar re-escribiendo una y otra vez mis *passwords* para hacer uso de las aplicaciones de mi empresa", y tampoco quieren tener que proporcionar múltiples credenciales para múltiples aplicaciones, lo cual un usuario diría así: "No quiero tener un usuario y password diferente para cada aplicación que tenga que usar".

En definitiva, se debe conseguir simplificar la experiencia de usuario de cara a identificarse en las aplicaciones. A esto se le conoce como 'Identificación-Única' o '*Single Sing-on*'.

Un ejemplo claro y muy conocido de '*Single Sing-on*', es el proporcionado por el Directorio Activo de Windows. Cuando tu usuario pertenece a un Dominio, se escribe la *password*/clave solamente una vez al principio del día (cuando se enciende el ordenador) y esta identificación da acceso a los diferentes recursos de la red interna, como impresoras, servidores de ficheros, *proxy* para salir a Internet, etc. Sin duda alguna, si cada vez que se accede a alguno de los recursos anteriores tuviéramos que

escribir nuestra *password*, no nos gustaría nada. Estamos acostumbrados a la transparencia proporcionada por la 'Autenticación Integrada de Windows'.

Irónicamente, la popularidad de *Kerberos* ha ido cayendo desde el punto de vista de que no ofrece, por sí solo, una solución real flexible e inter-organización a través de Internet. Esto es así porque el controlador de un Dominio contiene todas las claves a todos los recursos de una organización, y está también celosamente protegido por *firewalls*. Si se está fuera de la oficina, se nos exige normalmente conectarnos por VPN (*Virtual Private Network*) para acceder a la red corporativa. También, *Kerberos* es inflexible en términos de la información que proporciona. Sería muy útil poder extender el ticket de *Kerberos* para incluir **atributos arbitrarios (*Claims*)** como la dirección de correo electrónico o roles de aplicación, pero actualmente esto no es una capacidad de *Kerberos*.

A nivel genérico y sin ligarnos a ninguna plataforma, los *Claims* se diseñaron para proporcionar la flexibilidad que otros protocolos no tienen. Las posibilidades están limitadas solo por nuestra imaginación y las políticas de cada departamento de IT.

Los protocolos estándar para intercambiar *Claims* se han diseñado específicamente para cruzar fronteras de seguridad, como seguridades perimetrales, *firewalls* y diferentes plataformas, incluso de diferentes organizaciones. En definitiva, se pretende hacer más fácil la comunicación segura entre diferentes entornos y contextos.

***Claims*:**
Los *Claims* desacoplan las aplicaciones de los detalles de Identidad y Autenticación. Mediante esta aproximación, la propia aplicación no es ya responsable de autenticar a los usuarios.

Ejemplo de la vida real: Simplificando y para entender completamente el concepto de un Claim, vamos a compararlo con la vida real, donde estamos rodeados también de '*Claims*'. Una muy buena analogía es el "protocolo de autenticación" que seguimos cada vez que vamos al aeropuerto a tomar un vuelo.

No podemos simplemente llegar a la puerta de embarque enseñando nuestro DNI o Pasaporte (y mucho menos subir sin autorización, claro). En lugar de eso, se nos requiere ir a un punto intermedio/anterior (comparable a un 'Otorgador' o STS) donde tenemos que hacer *check-in* y facturar equipaje en su caso. En dicho mostrador se nos requiere que presentemos unas credenciales iniciales con sentido dependiendo de dónde viajamos (Similares a las credenciales de identidad utilizadas por la Organización, p.e. AD). Si viajamos dentro de la Unión Europea, necesitaremos como mínimo el DNI. Si viajamos con un viaje Internacional fuera de la UE, se nos requerirá el Pasaporte. Si se viaja con niños pequeños, también podemos dar sus nombres, los cuales quedan agregados a los datos de nuestro vuelo (Mas datos, añadimos otros tipos de '*Claims*'). Una vez verificadas nuestras credenciales iniciales (DNI/Pasaporte) simplemente mirando nuestra cara y comprobando que coincide con la foto del documento (Autenticación), y si todo está en orden, nos otorgarán una tarjeta de embarque válida exclusivamente para nuestro vuelo (Otorgamiento de *token* de seguridad y conjunto de *Claims* para mi aplicación).

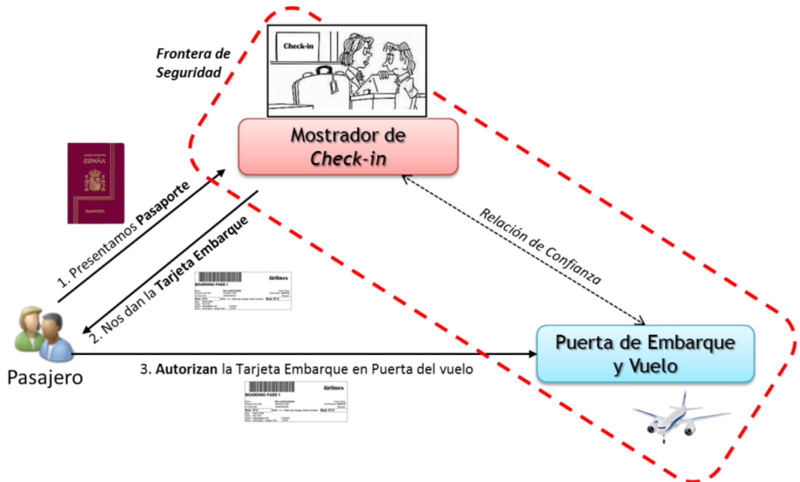


Figura 2.- Orientación a Claims, "cuando viajamos".

Una tarjeta de embarque proporciona mucha información. El personal de la puerta de embarque conocerá nuestro nombre, si somos un 'Viajero Frecuente' con una distinción especial (autorización en la aplicación y personalización), nuestro número de vuelo (nuestra aplicación) y nuestro número de asiento (autorización de acceso a un recurso que es el asiento). Y lo más importante de todo, una vez que pasamos la puerta de embarque (frontera de seguridad de nuestra aplicación), normalmente solo se nos requerirá de nuestra tarjeta de embarque (*token* de seguridad de aplicación con un conjunto de *Claims*) en las autorizaciones aplicadas en el avión.

También hay cierta información muy especial en la tarjeta de embarque y es que se encuentra codificada con un código de barras o banda magnética, lo cual prueba que el billete fue otorgado por una compañía aérea y no es una falsificación (Esto es comparable a una firma electrónica).

En esencia, una tarjeta de embarque es un conjunto de *claims* firmado, hecho por la aerolínea para nosotros. Declara que se nos permite embarcar a un vuelo en particular, en una hora concreta, y un asiento en particular.

También es interesante destacar que pueden existir diferentes formas de obtener la tarjeta de embarque (conjunto de *claims*). Podríamos haberlo sacado por Internet o en una máquina auto-servicio del Aeropuerto. Al personal de la puerta de embarque le da igual el método que utilizamos, simplemente nos autorizará a entrar.

En aplicaciones de software, **a dicho conjunto de *claims* otorgado por un STS, se le denomina**, como hemos adelantado antes, **TOKEN DE SEGURIDAD**. Cada token está firmado por el 'otorgador'/STS que lo creó.

Una aplicación basada en *Claims* considera a los usuarios ya autenticados si simplemente presentan un *Token* de seguridad válido otorgado por un STS en el cual confíe nuestra aplicación.

Traduciéndolo a un esquema de aplicación con seguridad orientada a Claims, el diagrama es muy parecido, pero cambian los elementos por componentes de software:

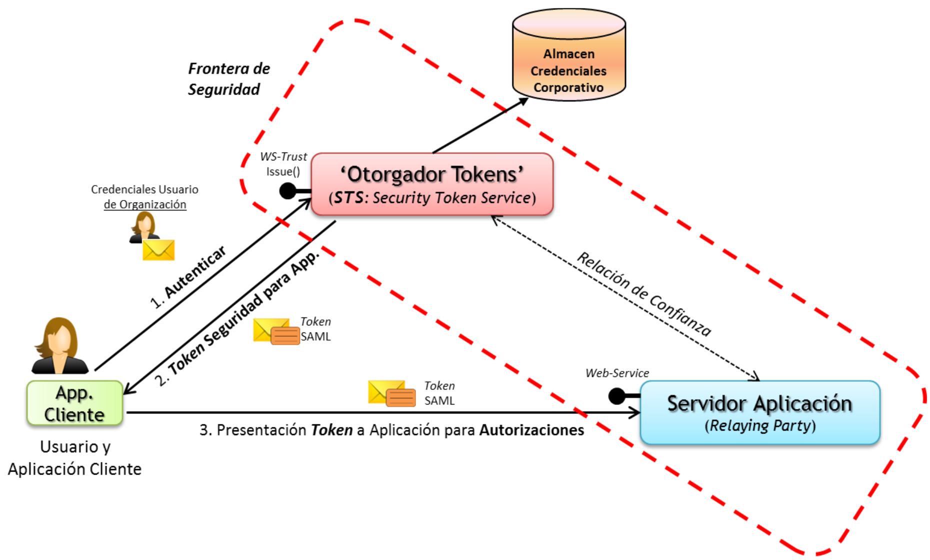


Figura 3.-

Todo lo que necesita nuestra aplicación es un *token* de seguridad proporcionado por el servicio 'otorgador' o emisor de *tokens* (técnicamente, a esta figura se le conoce como STS o *Security Token Service*), en el cual confía nuestra aplicación.

Nuestra aplicación no se romperá ni habrá que cambiar su diseño ni implementación si el departamento de IT decide actualizar la plataforma de seguridad e identidad, y por ejemplo se requiere ahora una 'smart-card' con un certificado X.509 en lugar de un nombre de usuario y password, o cualquier otro tipo diferente de identidad. Al tener nuestra aplicación desacoplada de la identidad inicial, no tendríamos que cambiarla en absoluto, ni re-codificación, re-compilación, ni tan siquiera una re-configuración.

Ventaja del Desacoplamiento de Credenciales

Para el desarrollo de aplicaciones, la ventaja es clara: La aplicación en sí no necesita preocuparte por qué tipo de credenciales presenta inicialmente el usuario. El departamento de IT de la organización/empresa habrá decidido eso. **Nuestra aplicación trabajará solamente** con el equivalente a la 'Tarjeta de Embarque', un ***token* de seguridad de la aplicación**, y el código a programar estará solamente relacionado con estos *tokens* de aplicación, sin importar que credenciales iniciales fueron presentadas por el usuario al STS (Kerberos-AD, Usuario-Password, Certificados, etc.).

No hay duda de que los Controladores de Dominio (en red Microsoft) u otro tipo de tecnologías (LDAP, etc.) seguirán guardando los recursos organizacionales. Y las

relaciones de confianza entre sistemas de identidad y autenticación seguirán dependiendo de aspectos políticos. La identidad orientada a *Claims* no va a cambiar nada de eso. Sin embargo, al posicionar una capa de *claims* sobre nuestros sistemas y realizar dicho desacoplamiento, podremos conseguir mucho mejor el objetivo final de una solución '*Single Sing-on*' que incluso esté abierta a cualquier tecnología de Identificación y Autenticación (no solo Directorio Activo de Microsoft, también cualquier otro directorio LDAP, Certificados, repositorios de usuarios etc.).

***NOTA*:**
Los *Claims* se integran con sistemas existentes de seguridad para permitir una mayor compatibilidad entre dichos sistemas y aplicaciones seguras y en definitiva, reducir obstáculos técnicos. **La 'Orientación a *Claims*' abre la compatibilidad de nuestras aplicaciones hacia cualquier tipo de tecnologías de Identidad.**

Tipos de Arquitecturas orientadas a *Claims*

Dependiendo del tipo de aplicación que estemos implementando (Web, Cliente-Rico, etc.), la aproximación arquitectural será ligeramente diferente. Por ejemplo, en una aplicación Web visual tradicional (HTML basado en el navegador) tendrá una técnica ligeramente diferente a una aplicación N-Tier (*Rich* o *RIA*) en la forma en la que los *Claims* se comunican desde el 'otorgador' (STS) a la aplicación.

En definitiva, el objetivo de estas arquitecturas es permitir una federación bien con un cliente rico (Rich) o con un navegador (IE o cualquier otro navegador).

- **Cliente Rico:** La federación con un cliente rico se basa en las especificaciones SOAP avanzadas de WS-*. Concretamente basándonos en ***WS-Trust*** y ***WS-Federation Active Requestor Profile***. Estos protocolos describen el flujo de comunicaciones entre clientes ricos (como aplicaciones cliente Windows) y servicios web (como Servicios WCF) para solicitar un token de seguridad al 'otorgador/emisor' (STS) y pasarle entonces dicho token al servicio web de la aplicación, para que realice la autorización.

- **Cliente Web**: La federación con un cliente Web está basada en ***WS-Federation Pasive Requestor Profile***, que describe un flujo de comunicación similar entre el navegador y la aplicación Web en el Servidor. En este caso, se basa en redirecciones del navegador, HTTP GET y HTTP POST para solicitar y pasar *tokens* de seguridad.

Tabla 2.- Guía de Arquitectura Marco

Regla Nº: D30.	**Hacer uso de 'Seguridad Orientada a *Claims*' como sistema preferido para aplicaciones empresariales complejas que deban integrarse a sistemas de Identidad corporativos.**

○ **Normas**

- Las aplicaciones empresariales complejas normalmente deben integrarse de forma transparente en los sistemas corporativos de Identidad de usuarios. Esto significa hacer uso de un '*single sign-on*' que disponga la organización y no obligue a los usuarios a tener otras credenciales diferentes para nuestra aplicación.

Ventajas del uso de Seguridad orientada a *Claims*

→ Transparencia y propagación de las credenciales corporativas

→ Desacoplamiento de la Aplicación con respecto al sistema de Identidad de usuarios

Referencias

- **A Guide to Claims–based Identity and Access Control**
  http://msdn.microsoft.com/en-us/library/ff423674.aspx

4.2.- Cache

El *Cache* puede incrementar drásticamente el rendimiento y grado de respuesta de una aplicación. Simplemente hay que conocer de forma muy precisa en qué puntos de la aplicación se puede o no se puede hacer uso de cache. También hay que tener en cuenta que un mal uso del cache puede por el contrario degradar el rendimiento y la escalabilidad de la aplicación.

Se debe hacer uso de cache para optimizar la referencia a búsquedas de datos, evitar comunicaciones remotas y en general evitar procesamientos duplicados. Cuando se

implementa cache, debemos decidir cuándo vamos cargar los datos en el cache y cuando se van a eliminar los datos caducados.

Es preferible intentar pre-cargar datos usados frecuentemente y hacerlo de una forma asíncrona o utilizando procesos '*batch*', que eviten retrasos al cliente.

Considerar las siguientes guías a la hora de diseñar el cache de una aplicación.

- **Situación del Cache**: Es fundamental elegir correctamente la situación del cache. Si la aplicación está desplegada en una Granja de Servidores Web (*Web-Farm*), evitar hacer uso de cache local a cada nodo de la granja (como las sesiones ASP.NET en el espacio de memoria de los procesos del Servicio Web), pues no podremos entonces balancear sin afinidad a dicho clúster-software. En lugar de eso, considerar el uso de cachés distribuidos y sincronizados entre los diferentes servidores de una forma automática.

- **Cache en formato preparado**: Considerar el uso de datos en un formato ya preparado, cuando se usa el cache. Por ejemplo, en lugar de cachear simplemente datos-texto simple, cachear objetos serializables que al mismo tiempo actúen como entidades.

- No hacer cache de datos muy volátiles y nunca cachear datos sensibles/críticos a menos que estén cifrados dentro del cache.

- Para operaciones con cierta duración y concatenación de sub-operaciones críticas, no depender de que existan ciertos datos en la cache, pueden haber sido eliminados. Implementar un mecanismo para gestionar fallos de cache, por ejemplo recargando el elemento de su fuente original, etc.

- Se debe tener especial cuidado cuando se accede a la cache desde múltiples *threads*. En ese caso, asegurarse de que todos los accesos al cache son '*thread-safe*' de forma que se mantenga la consistencia, utilizando mecanismos de sincronización.

Tabla 3.- Guía de Arquitectura Marco - Cache

Regla Nº: D31.	**Uso de CACHE en la aplicación**

○ **Normas**

- Se deberá usar la caché para el acceso continuo a datos estáticos o con datos

que no cambian constantemente.

- El acceso a un servidor de base de datos resulta costoso en cuanto a creación, acceso o transporte de dichos datos, usando la caché, mejorará este rendimiento.

4.3.- Gestión de Configuración

El diseñar un mecanismo de configuración apropiado es importante para la seguridad y la flexibilidad de nuestra aplicación. Si no se realiza correctamente nuestra aplicación podrá ser vulnerable a diferentes ataques y también puede generar sobrecarga de trabajo innecesario a los administradores de la aplicación. Considerar las siguientes guías generales relativas a la gestión de la Configuración:

- Considerar cuidadosamente qué características deben poder ser configuradas externamente. Verificar que realmente hay una necesidad de negocio para cada característica configurable y simplificar exponiendo las mínimas opciones posibles de configuración. Una complejidad excesiva de la configuración puede dar lugar a sistemas excesivamente complejos de administrar y mantener lo cual puede provocar mal funcionamiento e incluso agujeros de seguridad por una configuración incorrecta.

- Decidir si la configuración se guardará centralmente o si se aplicará a los usuarios en el momento de arrancar la aplicación (por ejemplo basado en políticas de Directorio Activo). Considerar como se restringirá el acceso a la información de configuración y hacer uso de procesos ejecutándose con los mínimos privilegios posibles mediante cuentas de servicio correctamente configuradas.

- Cifrar información sensible en el almacén de configuración. Por ejemplo, partes cifradas dentro de un fichero .config.

- Categorizar los elementos de configuración agrupados en diferentes secciones lógicas dependiendo del uso de cada configuración.

- Categorizar también los elementos de configuración en secciones lógicas si la aplicación tiene varios niveles físicos (*Tiers*). Si el servidor de aplicaciones se ejecuta en un '*Web-Farm*', decidir qué partes de la configuración son compartidas y cuales son específicas para cada

nodo/máquina en la que la aplicación se ejecuta. Entonces elegir un almacén apropiado para cada sección.

- Proporcionar un Interfaz de Usuario para los Administradores mediante el cual puedan editar la información de configuración (Aplicación tipo Snap-in sobre la MMC, de edición de configuración, por ejemplo).

4.4.- Gestión de Excepciones

El diseñar una buena estrategia de gestión de excepciones es importante de cara a la seguridad y estabilidad de la aplicación. Si no se realiza correctamente, puede ser muy complicado diagnosticar y resolver los problemas de una aplicación. También puede dejar a la aplicación vulnerable a ataques como DoS (*Denial of Service*) y mostrarse información sensible procedente de las excepciones/errores internos.

Una buena aproximación es diseñar un mecanismo centralizado de gestión de excepciones y considerar el proporcionar puntos de acceso al sistema de gestión de excepciones (como eventos WMI) para soportar sistemas de monitorización a nivel empresarial como 'Microsoft System Center'.

Considerar las siguientes guías generales sobre la gestión de excepciones:

- Diseñar una estrategia apropiada de propagación de excepciones que envuelva o reemplace las excepciones (errores internos), o añada información extra según se requiera. Por ejemplo, permitir que las excepciones suban hacia las capas superiores hasta llegar a las 'capas frontera' (como Servicios-Web o Capa de Presentación Web ASP.NET), donde dichas excepciones serán registradas (*logs*) y transformadas según sea necesario antes de pasarlas a la siguiente capa (normalmente, antes de que lleguen a la capa de presentación o interfaz gráfico de usuario).

- Considerar incluir un identificador de contexto de forma que las excepciones relacionadas puedan asociarse a lo largo de diferentes capas y poder identificar cual es la causa raíz de los errores y faltas. También asegurarse de que el diseño tiene en cuenta las excepciones no gestionadas (*unhandled exceptions*).

- No hacer 'Catch()' de excepciones/errores internos a menos que se gestionen o se quiera añadir más información.

- Nunca hacer uso de excepciones para controlar el flujo de la aplicación.

- Diseñar una estrategia adecuada de registro (*logging*) y notificación de errores críticos que muestre la suficiente información sobre el problema, de forma que permita a los administradores de la aplicación el poder recrear el escenario, pero que al mismo tiempo no revele información confidencial al

usuario final (en los mensajes que llegan al usuario ni tampoco en los mensajes registrados en los *logs*).

4.5.- Registro/Logging y Auditorías

El diseñar una buena estrategia de Logging y de Instrumentalización es importante de cara a la seguridad y diagnósticos de la aplicación. Si no se realiza correctamente, la aplicación puede ser vulnerable a amenazas de repudio, donde los usuarios niegan sus acciones y los ficheros/registros de log pueden requerirse para procedimientos legales que pretendan probar sus acciones. Se debe auditar y registrar la actividad de la aplicación en las diferentes capas en puntos clave que puedan ayudar a detectar actividades sospechosas y proporcionar pronto indicaciones de ataques serios. Las Auditorías son consideradas mejor autorizadas si son generadas en el preciso momento que se accede a los recursos y por los mismos algoritmos que acceden al recurso (AOP sería más transparente pero peor considerado de cara a Auditorías, porque no es tan claro el cómo y cuándo se está realizando dicha auditoría). Considerar las siguientes guías generales:

- Diseñar un sistema centralizado de registro/*logging* que capture los eventos más críticos de negocio. Evitar hacer un registro por defecto demasiado granularizado (generaría demasiado volumen de operaciones), pero considerar la posibilidad de cambiar la configuración en tiempo de ejecución y que entonces si se genere un mayor detalle de registro.

- Crear políticas de seguridad de gestión registros/*logs*. No guardar información sensible de accesos no autorizados, en los ficheros de log. Considerar como se accederá y pasarán datos de registro y auditoría de una forma segura entre las diferentes capas.

- Considerar el permitir diferentes tipos de trazas (*trace listeners*), de forma que pueda ser extensible a otros tipos de ficheros o registros, incluso modificable en tiempo de ejecución.

4.6.- Instrumentalización

La Instrumentalización puede implementarse mediante contadores de rendimiento y eventos que proporcionen a los Administradores información sobre el estado, rendimiento y salud de una aplicación.

4.7.- Gestión de Estados

La Gestión de estados (Sesiones, etc.) está relacionada con la persistencia de datos que representa el estado de un componente, operación o paso en un proceso. Los datos de estado pueden persistirse en diferentes formatos y de múltiples formas. El diseño de un mecanismo de gestión de estado puede afectar al rendimiento de la aplicación; el mantenimiento incluso de pequeños volúmenes de información de estados puede afectar negativamente al rendimiento y la habilidad de la aplicación de poder escalar correctamente. Solo deberían persistirse datos que realmente se necesitan persistir y se deben conocer todas las posibilidades de gestión de estado. Considerar las siguientes guías globales sobre la gestión de estados:

- Mantener la gestión de estado tan 'limpia' como sea posible; persistir la mínima cantidad de datos requeridos para mantener estado.
- Asegurarse de que los datos de estado son serializables si se necesitan persistir o compartir entre diferentes procesos y fronteras de red.
- Elegir un almacén apropiado de estados. El guardar estados en el espacio de memoria de un proceso es la técnica que mejor rendimiento puede ofrecer, pero solo si el estado no tiene que sobrevivir al proceso o a re-inicios del servidor. Persistir los estados a disco local o a Base de Datos si se quiere disponer de dichos estados después de que muera un proceso o incluso después de re-iniciar el/los Servidores.
- A nivel de tecnología a utilizar a la hora de compartir estados entre diferentes servidores, probablemente las dos más potentes son:
    - Hacer uso de un sistema de Cache/Estados que soporten Web-Farms y sincronización automática de los datos de dicho cache entre los diferentes servidores.
    - Hacer uso de un almacén central basado en una Base de Datos, si bien, esta opción baja el rendimiento al tener los datos persistidos físicamente.

4.8.- Validación

El diseñar un sistema de validación de datos de entrada es fundamental para la usabilidad y estabilidad de la aplicación. Si no se realiza correctamente, podemos dejar

a nuestra aplicación con inconsistencias de datos, violaciones de reglas de negocio y una experiencia de usuario pobre. Adicionalmente, puede dejar agujeros de seguridad como ataques '*Cross-Site Scripting*', ataques de inyecciones SQL, etc.

Desafortunadamente no hay una definición estándar que pueda diferenciar entradas de datos válidas de entradas perniciosas. Adicionalmente, el como la aplicación haga uso de los datos de entrada influenciará completamente los riesgos asociados a la explotación de una vulnerabilidad.

Considerar las siguientes guías globales de cara al diseño de Validaciones de entrada de datos:

- "*Todas las entradas de datos pueden ser perniciosas, mientras no se demuestre lo contrario*".

- Validar entradas de datos en cuanto a longitud permitida, formato, tipos de datos y rangos permitidos.

- **Lista opciones Permitidas vs. Lista de bloqueos**: Siempre que sea posible, diseñar el sistema de validación para permitir una lista que defina específicamente qué es aceptable como entrada de datos, en lugar de tratar de definir qué no es aceptable o puede comprometer al sistema. Es mucho más fácil abrir posteriormente el rango de alcance de una lista de valores permitidos que disminuir una lista de bloqueos.

- **Validación en Cliente y Servidor**: No confiar solamente en validaciones de entrada de datos exclusivamente en el lado cliente. En lugar de eso, hacer uso de validaciones cliente para dar al usuario una pronta respuesta y mejorar así la experiencia de usuario. Pero siempre se debe implementar también validación en el lado servidor para comprobar entradas de datos incorrectas o entradas perniciosas que se hayan 'saltado' la validación en la capa cliente.

- Centralizar la aproximación de validaciones en componentes separados si la lógica puede reutilizarse, o considerar el uso de librerías de terceras partes. De esta forma, se aplicarán mecanismos de validación de una forma consistente y homogénea a lo largo de la aplicación.

- Asegurarse de que se restringe, rechaza y/o limpian las entradas de datos del usuario

5.- IMPLEMENTACIÓN EN .NET DE ASPECTOS TRANSVERSALES

5.1.- Implementación en .NET de Seguridad basada en 'Claims'

La implementación en .NET se realiza con varias nuevas tecnologías de desarrollo y de infraestructura. La base principal de desarrollo (dentro de .NET Framework) es un nuevo pilar en .NET denominado **WIF** (*Windows Identity Foundation*), en nombre beta llamado 'GENEVA FRAMEWORK', que nos proporciona el API necesario para trabajar con los *tokens* de seguridad de la aplicación y sus conjuntos de *claims* internas. Este API incluso nos da la oportunidad de crear también nuestros propios STS (*Security Token Service*). Sin embargo, esto último no será necesario la mayoría de las veces, pues a nivel de Infraestructura, Microsoft ya nos proporciona un STS finalizado y listo para usar (el denominado **ADFS 2.0**) que trabajará en última instancia contra el Directorio Activo de Windows. Pero si queremos autenticar contra otros almacenes de credenciales, siempre podremos crearnos nuestro propio STS con el API de WIF.

5.1.1.- STS y ADFS 2.0

Como adelantábamos antes, nosotros podemos desarrollar con WIF nuestro propio STS para que de soporte a 'n' aplicaciones. Sin embargo, la mayoría de las veces lo más efectivo es hacer uso de un STS que sea un producto terminado.

Si se dispone de ***Windows Server 2008 R2 Enterprise Edition***, entonces podemos hacer uso de un nuevo servicio en Windows Server que es un STS. Este servicio se le ha denominado **Active Directory Federation Services (ADFS) 2.0**, que nos proporciona la lógica para autenticar al usuario contra el Directorio Activo y se puede personalizar cada instancia de ADFS para autenticar contra KERBEROS, FORMS-AUTHENTICATION o CERTIFICADOS X.509, pero siendo el almacén final de usuarios el propio *Windows Active Directory* (AD).

También se puede solicitar al STS de ADFS que acepte un *token* de seguridad de otro 'otorgador' (STS) perteneciente a otro sistema u autoridad (*realm*). A esto se le conoce como **Federación de Identidades** y es como se consigue single *sign-on* entre diferentes infraestructuras independientes.

El siguiente esquema muestra las tareas que realiza el STS, en este caso, ADFS 2.0 de Windows Server.

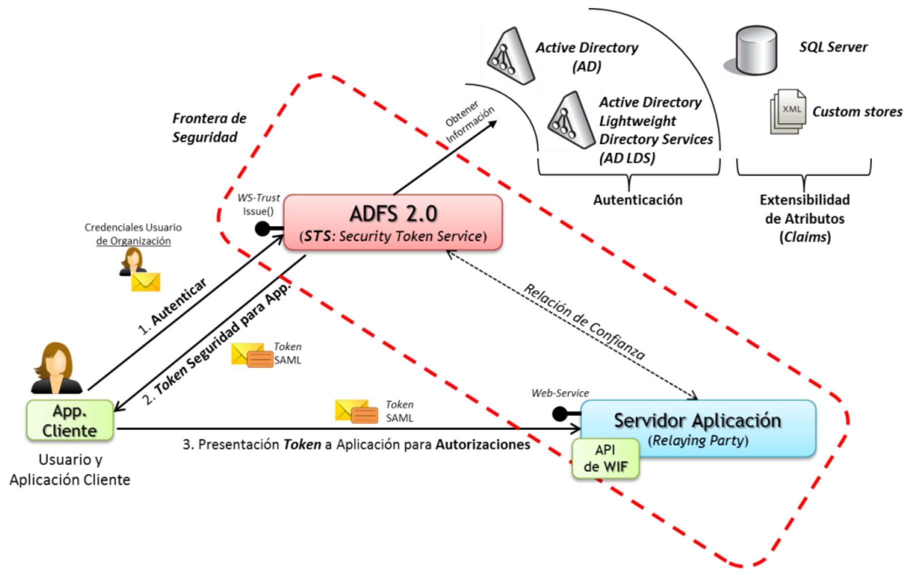


Figura 4.- Tareas que realiza el STS

Si se requiere que el almacén de credenciales sea otro diferente a *Windows Active Directory* o *Windows Active Directory Lightweight Directory Services* (este último es la evolución del antiguo ADAM, o *AD Application Mode*), entonces tendríamos que utilizar otro STS diferente a ADFS 2.0, bien desarrollándolo con WIF o bien consiguiendo algún otro STS del mercado.

Con ADFS 2.0, una vez que el usuario se ha autenticado, el ADFS crea *claims* sobre el usuario (las *claims* pueden estar definidas en el propio AD o también como atributos sobre SQL Server u otros almacenes *custom*). Finalmente el ADFS otorga el *token* de seguridad para nuestra aplicación, que incluirá un conjunto de *claims*.

ADFS tiene un motor de reglas que simplifica extraer los atributos LDAP de AD o AD-LDS. También permite añadir reglas que incluyan sentencias SQL de forma que se puedan extraer datos de usuario desde una base de datos en SQL Server con atributos de usuario extendidos. Otra opción es realizar esta extensibilidad de atributos con otros almacenes propios XML. Esta extensibilidad de atributos en almacenes externos a AD es fundamental porque muchas veces los datos de los usuarios en las organizaciones, están fragmentados. ADFS esconde dicha fragmentación. Además, si necesitamos añadir atributos/*claims*, en las grandes organizaciones con políticas más restringidas, es mucho más factible hacerlo a nivel de un almacén externo que solicitar a IT que extienda el esquema de los datos de usuario en el Directorio Activo.

Además, gracias a esta composición de datos de usuario, si se decide cambiar de lugar el almacén de ciertas propiedades de usuario, esto será completamente transparente desde ADFS 2.0 hacia afuera.

Las aplicaciones con seguridad basada en *claims*, lógicamente, esperan recibir *claims* sobre el usuario (como roles o permisos de aplicación e incluso datos personales), pero **a nuestra aplicación no le importa de dónde vienen dichos *claims*,**

esta es una de las ventajas del desacoplamiento de la aplicación con la identidad, basados en un STS (ADFS 2.0 en este caso).

Arquitecturalmente, ADFS 2.0 está construido sobre el *framework* de WIF (*Windows Identity Foundation*) y WCF (*Windows Communication Foundation*).

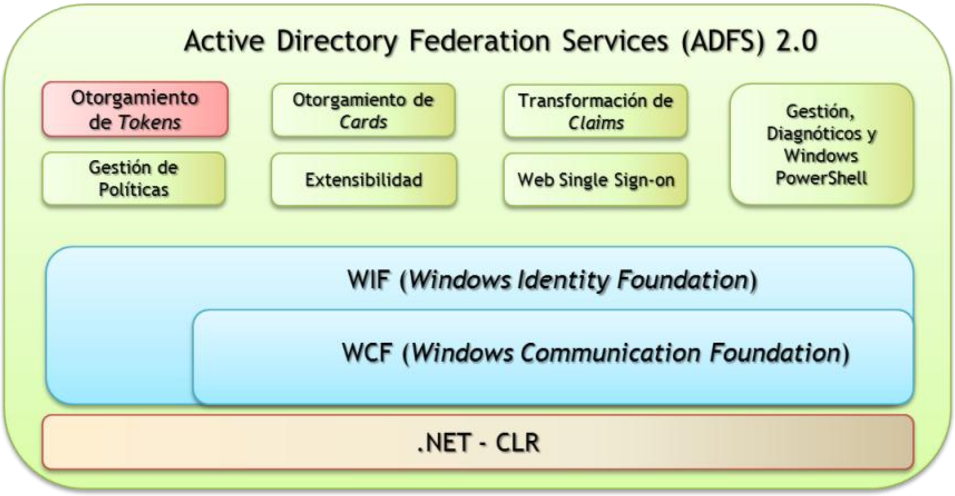


Figura 5.- Arquitectura de ADFS 2.0

Como parte fundamental de ADFS 2.0 está el STS (*Security Token Service*) que usa AD (Active Directory) como su almacén de identidades y LDAP, SQL Server o un almacén *custom* como almacén extendido de propiedades de usuarios.

El STS de ADFS 2.0 otorga *tokens* de seguridad basándose en varios protocolos y estándares, incluyendo WS-Trust, WS-Federation y SAML 2.0 (Security Assertion Markup Language 2.0). También soporta *tokens* en formato SAML 1.1.

ADFS 2.0 está diseñado con una clara separación entre los protocolos de comunicación y los mecanismos internos de otorgamiento de *tokens*. Los diferentes protocolos de comunicación se trasforman a un modelo de objetos estándar en la entrada del sistema mientras que internamente ADFS 2.0 utiliza el mismo modelo de objetos para todos los protocolos. Esta separación o desacoplamiento permite a ADFS 2.0 ofrecer un modelo muy extensible, independiente de las peculiaridades de cada protocolo, como se puede apreciar en el diagrama.

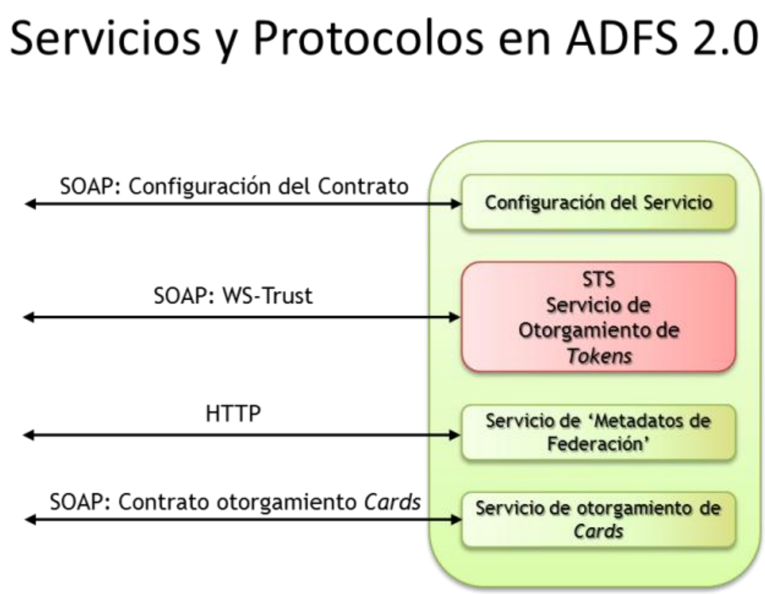


Figura 6.- Pasos para implementar en nuestra aplicación la Identidad basada en Claims

5.1.2.- Pasos para implementar 'Orientación a Claims' con WIF

Hay una serie de pasos que normalmente deberemos hacer para implementar 'Orientación a Claims' con WIF:

Paso 1 – Añadir a nuestra aplicación lógica que soporte claims

Nuestra aplicación necesita poder validar los token de seguridad entrantes y como extraer los *claims* que tiene dentro el *token*. Para ello, WIF (*Windows Identity Foundation*) nos proporciona un API y modelo de programación para trabajar con las claims que puede utilizarse tanto desde servicios **WCF** (*Windows Communication Foundation*) como desde aplicaciones Web **ASP.NET**. Si por ejemplo, se es familiar con API de .NET Framework tal como **IsInRole()** o propiedades como **Identity.Name**, este nuevo API de WIF es muy similar y extiende el API de .NET, teniendo ahora una propiedad más a nivel de Identidad: **Identity.Claims**. Esta propiedad nos da acceso a las *claims* que fueron otorgadas por el STS (ADFS 2.0), identificándolas, y proporcionando también información sobre quien las otorgó y qué contienen.

Desde luego, hay mucho más que aprender sobre el modelo de programación de WIF, pero por ahora solo recordar que debemos referenciar al assembly de WIF

(Microsoft.IdentityModel.dll) desde nuestro servicio WCF o aplicación ASP.NET para poder hacer uso del API de WIF.

Paso 2 – Adquirir o construir nuestro 'Otorgador de tokens' (STS)

Para una gran mayoría de escenarios, la opción más segura y rápida será hacer uso de ADFS 2.0 como **'Otorgador de *tokens*' (STS)**. Si ADFS 2.0 no cumple nuestros requerimientos de autenticación (por ejemplo, se requiere autenticar contra almacenes de identidad diferentes a AD), podremos crear nuestro propio STS con el API de WIF, pero el construir un STS con una alta calidad de 'Producción' es lógicamente más complicado que el uso de WIF en nuestra aplicación, por lo que a menos que se disponga de un buen nivel de experiencia en aspectos avanzados de seguridad, es recomendable adquirir un STS del mercado.

En cualquier caso, es perfectamente factible desarrollar con WIF un *STS-Custom*, y en ciertos casos puede merecer la pena el esfuerzo, teniendo en cuenta que un STS es reutilizable como base para 'n' aplicaciones consumidoras. No es solo para una única aplicación.

Finalmente, destacar que ADFS 2.0 permite ser personalizado mediante varios puntos de extensibilidad, como adición de atributos/*claims* en almacenes externos tipo SQL Server o almacenes XML.

Paso 3 – Configurar la Aplicación para que confíe en el 'Otorgador de *tokens*' (STS – ADFS 2.0)

Para que pueda funcionar nuestra aplicación con los *tokens* de seguridad otorgados por el STS (ADFS 2.0, en este caso), es necesario establecer una relación de confianza entre ambos. La aplicación debe confiar en que el STS identificará, autenticará a los usuarios y creará las *claims* respectivas (Roles y/o datos personales).

Hay varios puntos a tener en cuenta sobre el **'Otorgador de *tokens*' (STS)** cuando vamos a establecer la relación de confianza:

- ¿Qué *claims* ofrece el **'Otorgador de *tokens*'** (STS)?
- ¿Qué clave debe de utilizar la aplicación para validar las firmas de los *tokens*?
- ¿Qué URL deben acceder los usuarios para poder solicitar un *token* al STS?

Los *claims* pueden ser cualquier dato que nos imaginemos sobre un usuario, pero a nivel práctico, lógicamente hay algunos *claims* típicos. En definitiva se tiende a ofrecer piezas comunes de información, como **nombre**, **apellidos**, **correo electrónico**, **roles/grupos de aplicación**, etc.

Se puede configurar cada **'Otorgador de *tokens*'** (STS) para que ofrezca diferente número y tipos de *claims*, por lo que se puede ajustar a las necesidades de una aplicación, y viceversa, ajustar la aplicación a los *claims* ya pre-establecidos por la organización/empresa en sus STS corporativos.

Todas las preguntas anteriores se pueden responder ‘preguntando’ al STS por los METADATOS DE FEDERACION (*federation metadata*), que es en definitiva un documento XML que proporciona el STS a la aplicación. Incluye una copia serializada del certificado del STS que proporciona a la aplicación la clave pública correcta para que pueda verificar los *tokens* entrantes. También incluye un lista de los claims que ofrece el STS, la URL donde la aplicación cliente obtendrá el token y otros detalles técnicos, como el formato del *token* (**SAML** normalmente). WIF dispone de un asistente que configura automáticamente las propiedades de identidad de las aplicaciones basándose en estos metadatos. Solo necesitamos proporcionar a dicho asistente la URL del STS y con ello obtendrá los metadatos y configurará apropiadamente nuestra aplicación.

Paso 4 – Configurar al ‘Otorgador de *tokens*’ (STS – ADFS 2.0) para que reconozca a nuestra aplicación

El STS también necesita conocer algunos datos sobre nuestra aplicación antes de que pueda otorgar cualquier token.

- ¿Qué URI (*Uniform Resource Identifier*) identifica a la aplicación?
- De los claims que ofrece el STS, ¿Cuáles requiere obligatoriamente la aplicación y cuales son opcionales?
- ¿Deberá el STS cifrar los tokens? Si es así, ¿Qué clave debe utilizar?

Cada aplicación es diferente y no todas necesitan los mismos *claims*. Una aplicación puede necesitar conocer los grupos o roles de aplicación, mientras que otra puede solo necesitar el nombre y apellidos. Por lo que cuando un cliente solicita un *token*, parte de dicha petición incluye un identificador de la aplicación que está tratando de acceder. Ese identificador es un URI y, en general, lo más sencillo es utilizar la URL de la aplicación o servicio-web, por ejemplo, *http://www.miempresa.miapp*.

Si la aplicación que estamos construyendo tiene un grado razonable de seguridad, probablemente se haga uso de SSL (HTTPS) tanto para el STS como para la propia aplicación. Eso protegerá a toda la información en general durante su comunicación.

Si la aplicación tiene requerimientos de seguridad aún más fuertes, también pueden pedir al STS que los *tokens* estén cifrados, en cuyo caso, la aplicación tendrá su propio certificado (y clave privada). El STS necesitará tener una copia de dicho certificado (sin la clave privada, solo con la clave pública), para poder cifrar los *tokens* otorgados a los usuarios de la aplicación.

Una vez más, los ‘metadatos de federación’ nos facilita este intercambio de información. **WIF** incluye una herramienta llamada **FedUtil.exe** que genera un documento de ‘*metadatos de federación*’ para nuestra aplicación de forma que no tengamos que configurar manualmente al STS con todas estas propiedades.

5.1.3.- Beneficios de la 'Orientación a *Claims*', WIF y ADFS 2.0

Los *claims* desacoplan la autenticación de la autorización de forma que la aplicación no necesita incluir la lógica de un modo específico de autenticación. También desacopla los roles de la lógica de autorización e incluso nos permite utilizar permisos más granularizados que lo que nos proporcionan los típicos roles/grupos de aplicación.

Podemos conceder accesos de seguridad a usuarios que anteriormente hubieran sido imposibles debido a que estuvieran en diferentes *Dominios/Forests*, no formaran parte de ningún dominio o incluso utilizasen un sistema de Identidad de otras plataformas y tecnologías no Microsoft (Esto último es posible mediante el uso de otros STS en lugar de ADFS 2.0).

Mejora la eficiencia de las tareas de IT al eliminar cuentas de usuario duplicadas a nivel de aplicaciones o Dominio y previene que el almacén de información crítica de los usuarios salga de las fronteras de seguridad de la organización (Sistemas controlados por IT).

5.2.- Implementación de Cache en plataforma .NET

El cache es algo que se puede implementar en diferentes niveles físicos e incluso situarlo en diferentes capas lógicas (capas). Pero la implementación en los diferentes niveles físicos suele ser muy diferente según si se implementa cache en el cliente (aplicaciones *Rich* y RIA) o en el servidor de aplicaciones.

5.2.1.- Implementación de Cache-Servidor con Microsoft AppFabric-Cache

Las aplicaciones escalables (aplicaciones Web y *N-Tier*) normalmente disponen de una arquitectura N-Capas como la presentada en esta guía, y finalmente la mayoría de las entidades y datos persistentes están almacenados en fuentes de datos que son la mayoría de las veces, bases de datos. Este tipo de aplicaciones permite escalar horizontalmente a nivel de servidores web y de componentes de negocio (mediante *web-farms*), e incluso podríamos extender el mismo paradigma de arquitectura no solamente en servidores tradicionales propios, sino también a otro contexto como 'Cloud-Computing' sobre Windows Azure, etc.

Sin embargo, una arquitectura lógica N-Capas con ciertos niveles físicos (*N-Tier*), tiene puntos críticos de cara a la escalabilidad y normalmente, el más crítico es el Servidor de Base de Datos relacional (SQL Server o cualquier otro SGBD). Esto es así porque el Servidor de Bases de datos normalmente **solo puede escalar de forma vertical** (mayor servidor, aumento de procesadores y memoria), pero los SGBD relacionales no pueden normalmente escalar de forma horizontal tanto para lectura como escritura, porque son sistemas orientados a conexión (p.e. en SQL Server, una conexión TCP en el puerto 1433).

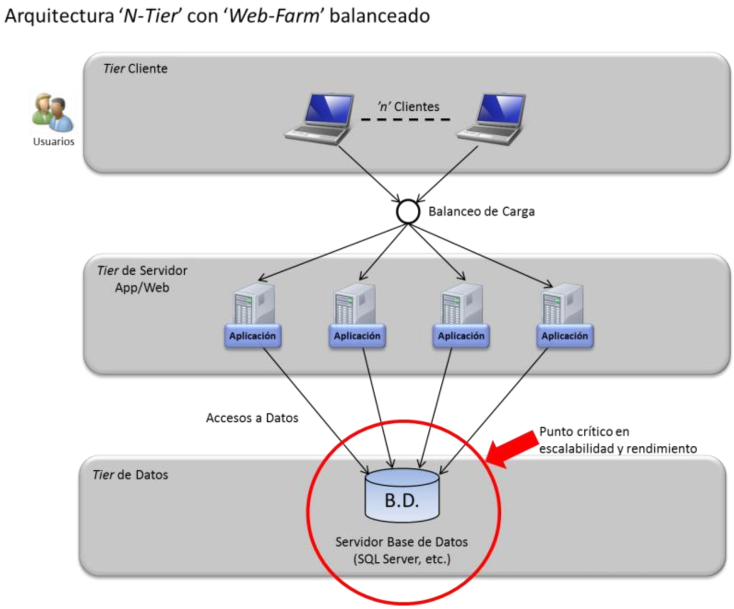


Figura 7.- Arquitectura 'N-Tier' con 'Web-Form' balanceado

En definitiva, normalmente cada vez que se carga una página web o se recarga un control de una aplicación Rich/RIA se tiene que acceder una serie de veces a la base de datos. Según la carga de la aplicación crece (aumenta el número concurrente de usuarios), esta frecuencia de operaciones en base de datos puede ofrecer serias limitaciones de escalabilidad e incluso de cuello de botella en el rendimiento debido a un gran incremento de contención de recursos, a las limitaciones físicas de obtener datos persistidos en disco, (Base de datos) y la latencia de consultas remotas al servidor de base de datos. En definitiva, el punto más débil en esta arquitectura, a la hora de escalar de forma sensible, es el servidor de base de datos (podríamos tener un clúster hardware de B.D. pero esto al fin y al cabo solamente nos ofrece una alta disponibilidad, no una mayor escalabilidad).

Si la aplicación hace uso de aproximaciones tradicionales de cache en el nivel Web (p.e. Sesiones de ASP.NET) para reducir presión contra la base de datos, entonces aparece un segundo reto, pues las sesiones de ASP.NET en cache-memoria, están ligadas a cada servidor, por lo que entonces hay que recurrir a trucos como que el balanceo de carga tiene que ser con afinidad, para ligar a los usuarios al servidor con el que inicialmente contactaron. Esto hace que el balanceo no sea el mejor, y potencialmente pueden aparecer desproporciones en el balanceo de carga. Además, estas sesiones en memoria no dispondrían de alta disponibilidad. Si uno de los servidores 'se cayera', sus sesiones ASP.NET se perderían.

Por último, se puede disponer de un único servidor de sesiones ASP.NET, pero esto significa tener un único punto de fallo. Y por último, si se persisten dichas sesiones en

la base de datos, entonces volvemos al problema inicial de menor rendimiento y sobrecarga del acceso a base de datos.

Necesitamos disponer de un cache en la memoria de los servidores Web/Aplicación, pero que sea un cache distribuido, es decir, sincronizado entre los diferentes servidores de una forma automática e inmediata.

Microsoft AppFabric-Cache (el nombre beta era '*Velocity*') proporciona un cache distribuido y en memoria, lo cual permite crear aplicaciones escalables, altamente disponibles y con un gran rendimiento. Desde el punto de vista de su uso, AppFabric-Cache expone una vista unificada de memoria distribuida a ser consumida por la aplicación cliente (en este caso, consumido el cache por las capas N-Layer de aplicaciones web ASP.NET o aplicaciones N-Tier con servicios WCF).

Mediante ***AppFabric-Cache*** las aplicaciones pueden mejorar drásticamente su rendimiento, pues estamos 'acercando' los datos a la lógica que los consume, (Capas de aplicación *N-Layer*) y por lo tanto reduciendo la presión ejercida contra el nivel físico del servidor de base de datos.

El *clúster* de ***AppFabric-Cache*** ofrece alta disponibilidad para evitar pérdidas de datos en las aplicaciones y simultáneamente incrementar la escalabilidad de la aplicación. La gran ventaja de este tipo de cache distribuido es que puede crecer su escalabilidad de forma flexible, simplemente añadiendo más servidores de cache.

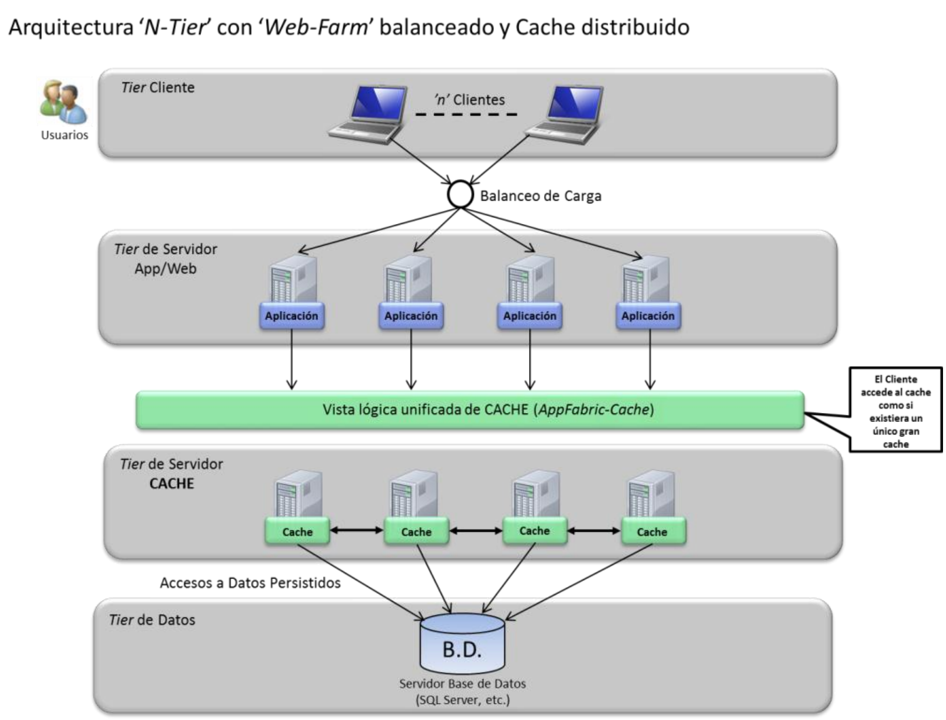


Figura 8.- Arquitectura 'N-Tier' con 'Web-Form' balanceado y Cache distribuido

La arquitectura de ***AppFabric-Cache*** consiste en un anillo de servidores-cache que ejecutan el servicio Windows de Cache. Por otro lado, las aplicaciones cliente del cache, hacen uso de una librería cliente .NET para comunicarse con la 'Vista lógica unificada del cache'.

Así pues, ***AppFabric-Cache***, nos permite crear un nuevo nivel físico como podemos apreciar en el esquema (o solo lógico como también veremos posteriormente). Esto hace posible conseguir **nuevos niveles de escalabilidad**, rendimiento y disponibilidad. Más específicamente, un alto grado de escalabilidad puede conseguirse precisamente minimizando el acceso a los datos persistidos en la base de datos. Al añadir este nuevo nivel de flexibilidad en la **escalabilidad**, la arquitectura física se ve también libre ahora de tener que realizar un balanceo de carga con afinidad (como cuando se usan sesiones ASP.NET en memoria). Y relativo al **rendimiento**, este se ve también sensiblemente mejorado porque estamos 'acercando' los datos a la lógica (Servidor de aplicaciones), mejorando por lo tanto los tiempos de respuesta y los tiempos de latencia (el acceso a un servidor de base de datos siempre será más lento que el acceso a un cache en memoria).

Adicionalmente tenemos también **alta disponibilidad**, conseguida por la disponibilidad de un *clúster* con redundancia, así pues, mitigando pérdidas de datos así como picos de carga que pueda haber en el nivel físico de datos (*Cluster-Hardware* de Base de datos en el supuesto caso de caída de un nodo/servidor de base de datos).

Por supuesto, también podríamos tener una arquitectura más simplificada si no requerimos de tanta escalabilidad, ejecutando el servicio de cache en el propio nivel del 'web-farm' de web o 'web-farm' de servidor de aplicaciones, como se puede ver en el esquema.

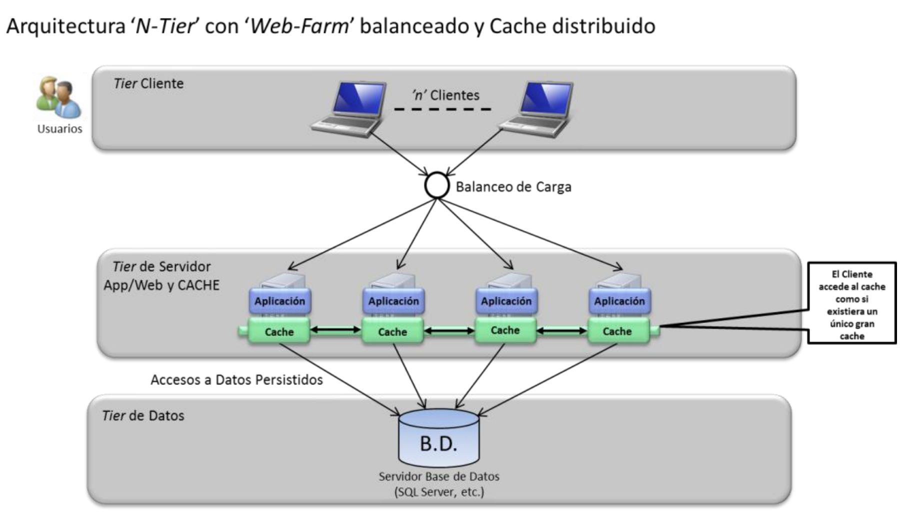


Figura 9.- Arquitectura 'N-Tier' con 'Web-Form' balanceado y Cache distribuido

Las aplicaciones cliente (Capas de nuestra Arquitectura N-Layer), pueden acceder al nivel de cache de ***AppFabric-Cache*** para guardar u obtener cualquier objeto CLR que sea serializable, mediante operaciones simples de tipo ‘add/put’ y ‘get’, como se muestra en el código fuente.

```
C#
//Standard AppFabric-Cache code

//Get default cache
DataCacheFactory _cacheFactory = new DataCacheFactory();
DataCache defaultCache = _cacheFactory.GetDefaultCache();

//Save object/value in cache
defaultCache.Put("Prod101", new Product("Libro N-Layer DDD"));

//Get data/value from cache
Product p = (Product) defaultCache.Get("Prod101");
```

Con este API mostrado se puede implementar un cache de servidor en una aplicación NLayer DDD separando cada tipo de objeto en las diferentes capas de la arquitectura (Capa de Infraestructura para clases Manager/Factory y Capa de Aplicación para uso explícito de cache (Put/Get) con operaciones concretas de entidades del dominio). Más adelante, en el presente capítulo, mostramos una aproximación de implementación de cache en una arquitectura NLayer DDD.

Clasificación de datos en AppFabric-Cache

Para poder hacer un buen uso del cache de AppFabric, es importante entender los tipos de datos que normalmente se cachean. Esos tipos de datos se pueden clasificar como **datos referencia, datos de actividad y datos de recursos**.

Los **datos Referencia** son datos que mayoritariamente utilizamos solo en modo lectura, como pueden ser datos de un *perfil de usuario, o datos de un catálogo de productos*. Este tipo de datos se actualiza de forma poco frecuente, por ejemplo, solo una vez al día o una vez a la semana. Sin embargo, los requerimientos de escalabilidad de estos datos referencia requieren normalmente de un gran número de peticiones de lectura contra esas pequeñas piezas de datos. Si se hiciera siempre contra la base de datos directamente, la escalabilidad quedaría muy limitada.

Por ejemplo, en un comercio electrónico, según aumente el número de usuarios que lo visita, el número de consultas al catálogo de productos puede aumentar drásticamente. Como normalmente los datos de los productos no cambian muy a menudo (el precio puede cambiar, pero no muy a menudo), este tipo de datos (catálogo de productos) es un muy buen candidato para ser incluido en el cache como datos referencia, lo que con gran seguridad aliviará muchísimo la carga de acceso que estuviera soportando el servidor de base de datos con respecto a si se consultara siempre el catálogo contra la base de datos.

Los **datos de tipo 'Actividad'** son datos que forman parte de las actividades de negocio y por lo tanto normalmente datos transaccionales. Un buen ejemplo en un comercio-e serían los datos de una '*cesta de la compra*'. Representan datos de actividad y por lo tanto estos datos normalmente solo se leen o se escriben. Después de la vida de una actividad (en este caso, cuando se va a pagar la compra), los datos de actividad se eliminan del cache y se persisten en la fuente de datos persistente (base de datos), en este ejemplo, los datos de la cesta de la compra se convertirían en un Pedido ya persistido finalmente en la base de datos. Anteriormente, si se hubiera hecho uso de sesiones ASP.NET para una 'cesta de la compra', el comercio-e hubiera requerido un balanceo de carga con afinidad, perjudicando parcialmente a la escalabilidad. Ahora, con AppFAbric-Cache, se puede almacenar la cesta de la compra en el cache distribuido, y el balanceo de carga puede ser puro, maximizando la escalabilidad de los servidores disponibles.

Los **datos de tipo 'Recurso'** son datos que son constantemente leídos y escritos, como un inventario de productos o un saldo de cuenta bancaria. Durante un proceso de pedido, los niveles de inventario pueden requerir ser monitorizados para asegurar niveles de stock. Sin embargo, según se procesan los pedidos, estos datos necesitan ser actualizados de forma concurrente para reflejar los cambios de stock. A veces, para mejorar el rendimiento y la escalabilidad, se relajan los niveles de coherencia de dichos datos. Por ejemplo, el proceso de pedidos puede sobre-vender artículos mientras en procesos separados se pueden estar generando compras o fabricación de nuevos artículos para volver a mantener los niveles de stock. Sin embargo, estos procesos conllevan ya más riesgos.

Jerarquía Lógica de Arquitectura de AppFabric-Cache

La jerarquía lógica de **AppFabric-Cache** está compuesta por **máquinas**, **hosts**, **caches nombrados**, **regiones** y **elementos de cache**. Las máquinas pueden ejecutar múltiples servicios de **AppFabric-Cache**, y cada servicio se considera un host de cache. Cada cache puede contener múltiples caches nombrados y dichos caches nombrados estarán a lo largo de las diferentes máquinas definidas en una configuración.

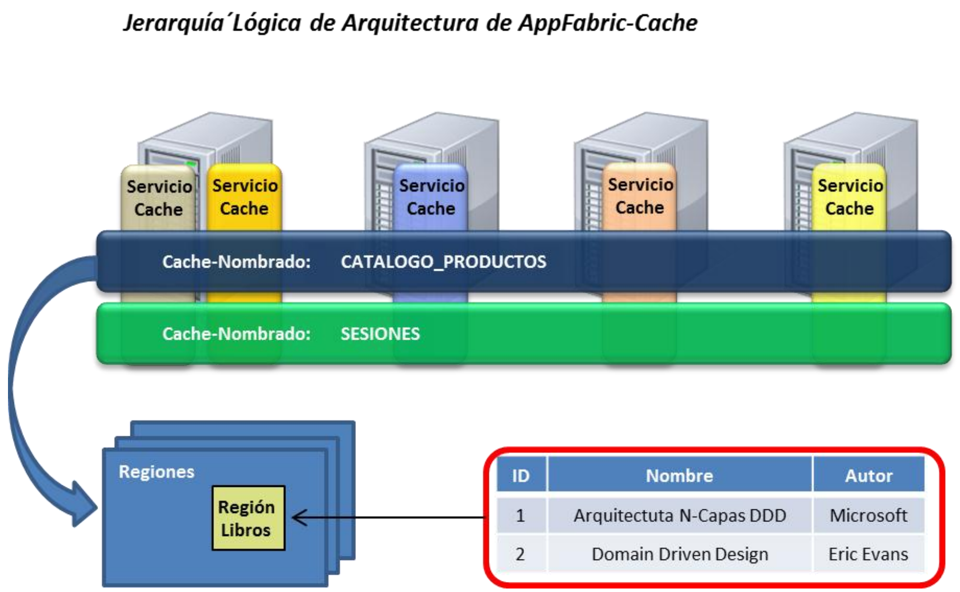


Figura 10.- Jerarquía Lógica de AppFabric-Cache

Cada cache-nombrado guarda un grupo lógico de datos, como SESIONES de usuario o un Catálogo de Productos. En los caches-nombrados también se establecen políticas sobre expiración de datos, disponibilidad, etc.

Las regiones explícitamente creadas (físicamente como contenedores de datos), pueden también existir dentro de cada cache-nombrado. Según se obtienen datos, las regiones se vuelven más útiles si la aplicación necesita direccionar un grupo de elementos de cache de forma conjunta. Sin embargo, la creación explícita de estas regiones es algo opcional. AppFabric-Cache creará implícitamente regiones por defecto si no se especifica una región de forma explícita.

Por último, dentro de las regiones (explícitas o implícitas), están los propios elementos de cache, que son responsables de mantener claves, objetos serializables, *tags*, *timestamps*, versiones y datos de expiración.

Otras implementaciones de Cache de Servidor

Por último, comentar que existen otras implementaciones de cache, como:

- *Memcached* de *Danga Interactive*
- Bloque de Cache de *Enterprise library de Microsoft P&P*

5.2.2.- *Implementación de AppFabric-Cache en aplicación ejemplo DDD NLayerApp*

No hay una única forma de implementar cache en una aplicación N-Capas DDD. Probablemente las opciones principales son:

- 1. **Uso explícito de cache en la Capa de Aplicación**: Esta opción deja muy claro qué servicios de aplicación y entidades del dominio relacionadas hacen uso de cache y cuáles no. Es probablemente la opción más quirúrgica a la par que clara, pues se utiliza el API de cache muy explícitamente en la Capa de Dominio.

- 2. **Uso de Repositorios de cache que sustituyan a repositorios normales**: Estos repositorios de cache, haciendo uso de IoC (Unity, etc.), pueden sustituir de forma transparente a repositorios estándar que accedan directamente a bases de datos (con EF, por ejemplo). Lo importante es que implementen y cumplan el mismo Interfaz de Repositorio. La ventaja es que puede cambiarse el uso de cache para ciertas entidades de una forma transparente e incluso 'en caliente' (XML de configuración de Unity) si se desea. Las desventajas son que debe utilizarse para todas las operaciones de un Repositorio (y agregado relacionado) y también que queda poco explícito para un revisor de código si se está o no utilizando cache, pues la única diferencia radica en la clase repositorio que se esté registrando en Unity como mapeo a la implementación del interfaz del repositorio. Este último punto (transparencia) puede verse como una desventaja o como una ventaja.

- 3. **Uso de AppFabric-Cache como cache de segundo nivel de Entity Framework**: Esta opción es la más transparente de todas pues podría hacer cache de todas las entidades que se utilicen con EF sin hacer nada en el código propio de nuestra aplicación. Es realmente una capa intermedia y transparente (infraestructura) entre nuestra aplicación y EF. Requiere del desarrollo de una librería/framework reutilizable, probablemente como algo externo al código de nuestra aplicación. De forma transparente, cuando una entidad se va a cargar desde la base de datos, el ORM primero comprueba si existe en el cache de segundo nivel, y en caso positivo se devuelven los datos desde el cache sin necesidad de consultar a la base de datos. Tanto NHibernate como Entity Framework pueden hacer uso de AppFabric-Cache como cache de segundo nivel. Para NHibernate existe un proyecto llamado nhibernate.caches.velocity (*sourceforge.net/projects/nhcontrib/files/NHibernate.Caches/*) y para Entity Framework EFCachingProvider by Jaroslaw Kowalski. (*code.msdn.microsoft.com/EFProviderWrappers*). En el futuro es posible que los propios ORMs decidan implementar esta funcionalidad.

- 4.- **Uso de AppFabric-Cache como *provider* para sesiones ASP.NET:** Este uso es complementario a los anteriores y se puede utilizar siempre que se

utilice ASP.NET. Esta nueva posibilidad elimina el problema de las sesiones ASP.NET en la memoria de servidores de *Web-Farms* balanceados de forma pura.

En nuestra implementación de AppFabric-Cache en la aplicación ejemplo NLayerApp, hemos optado por la primera opción por ser la más flexible, explicita y por lo tanto también la más educativa. Además, nos parece importante poder tratar con cache algunos aspectos/acciones de una entidad y no otros. Por ejemplo, podemos querer cachear las consultas/listas de una entidad y no cachear la consulta puntual de una instancia concreta de una entidad. Es probablemente, en definitiva, la forma más flexible, quirúrgica y explícita de hacer uso del *cache*.

Adicionalmente, y solo para el cliente ASP.NET, también podemos hacer uso de AppFabric-Cache para mantener las sesiones en su cache distribuido.

Implementación de clase manager de Cache en capa de Infraestructura

Siguiendo con las líneas de diseño de nuestra arquitectura, es importante que los elementos de infraestructura (en este caso uso de cache) estén localizados en la capa de infraestructura pero utilizados siempre mediante abstracciones (contrato/interfaz) declarados en la Capa del Dominio. De esta forma y mediante el uso de IoC e Inyección de Dependencias, podríamos llegar a sustituir la infraestructura de cache implementada de forma transparente y prácticamente sin impacto alguno (sustitución de implementación con AppFabric-Cache por otra tecnología futura). Adicionalmente este desacoplamiento permite también realizar pruebas unitarias contra mocks/stubs que realmente no estén haciendo uso del ***cache***.

A continuación se muestra el código simplificado de la clase **CacheManager**.

C# - Capa de Elementos Transversales de Infraestructura

Interfaz para abstracción e instanciación mediante contenedor IoC (Unity)

```
public sealed class CacheManager : ICacheManager,IDisposable
{
   DataCacheFactory _cacheFactory;

   /// Is recomended using "singleton" life time in the selected IoC
   public CacheManager()
   {
      //configuration for this cache factory is delegated in application
configuration file
      _cacheFactory = new DataCacheFactory();
   }

   public bool TryGet<TResult>(CacheItemConfig cacheItemConfig, out
TResult result)
   {
      if (cacheItemConfig != null)
      {
         //get default cache
         DataCache defaultCache = _cacheFactory.GetDefaultCache();

         string cacheKey = cacheItemConfig.CacheKey.GetCacheKey();
```

Obtenemos named-cache por defecto de AppFabric-Cache

```
            //get object from cache and check if exists
            object cachedItem = defaultCache.Get(cacheKey);

            if (cachedItem != null)      Intentamos obtener el valor solicitado desde el cache
            {
               result = (TResult)cachedItem;
               return true;
            }                    Si existe en cache, se devuelve. Si no, devolvemos un false
                    else
                    {
                        result = default(TResult);

                        return false;
                    }
                }
                else
                    throw new ArgumentNullException("cacheItem");
            }

            public void Add(CacheItemConfig cacheItemConfig, object
value)
            {
                if (value != null
                    &&
                    cacheItemConfig != null)
                {
                    //get default cache
                    DataCache defaultCache =
_cacheFactory.GetDefaultCache();

                    string cachekey =
cacheItemConfig.CacheKey.GetCacheKey();
                    TimeSpan expirationTime =
cacheItemConfig.ExpirationTime;

                    defaultCache.Put(cachekey, value,expirationTime);
                }
            }


            public void Dispose()
            {
                if (_cacheFactory != null)
                    _cacheFactory.Dispose();
            }

        }
```

Uso de clase manager de Cache desde capa de Aplicación

A continuación se muestra un ejemplo de implementación de clase de SERVICIO de Aplicación para controlar lo relativo a la entidad **Customer** cuando en algunos de sus métodos hacemos **uso del cache de AppFabric**:

```
C#

public class CustomerManagementService : ICustomerManagementService
{
    ICustomerRepository _customerRepository;
    ICountryRepository _countryRepository;
    ICacheManager _cacheManager;
```

Constructor con Dependencias requeridas

```
    public CustomerManagementService(ICustomerRepository
customerRepository, ICountryRepository countryRepository,ICacheManager
cacheManager)
        {
            _customerRepository = customerRepository;
            _countryRepository = countryRepository;
            _cacheManager = cacheManager;
        }
```

Lógica de Aplicación con **datos cacheados** para la entidad 'Customer'.

```
public List<Customer> FindPagedCustomers(int pageIndex, int pageCount)
     {
            //Implementing cache-aside pattern

            List<Customer> customerResults = null;
            CacheKey key = new CacheKey("FindPagedCustomers", new
{PageIndex=pageIndex,PageCount = pageCount });
            CacheItemConfig cacheItemConfig = new CacheItemConfig(key,
new TimeSpan(0, 0, 30));
```

Intentamos obtener los datos desde el **cache**.

```
            if (_cacheManager.TryGet<List<Customer>>(cacheItemConfig,
out customerResults))
                return customerResults;
            else
            {
                bool enabled = true;
                Specification<Customer> onlyEnabledSpec = new
DirectSpecification<Customer>(c => c.IsEnabled == enabled);
```

Si no existe en el **cache**, lo obtenemos de la B.D. y lo guardamos en la cache para futuras consultas

```
                customerResults =
_customerRepository.GetPagedElements(pageIndex, pageCount, c =>
c.CustomerCode, onlyEnabledSpec, true)
                                                     .ToList();
                _cacheManager.Add(cacheItemConfig, customerResults);
                return customerResults;
            }
     }
}
```

Uso de AppFabric Cache como provider de sesiones ASP.NET

Se puede utilizar siempre que se utilice ASP.NET. Esta nueva posibilidad de utilizar AppFabric Cache con las sesiones ASP.NET, elimina el problema de las sesiones ASP.NET en la memoria de servidores de *Web-Farms* balanceados de forma pura.

Este problema se produce cuando entra una petición http balanceada. Ahí normalmente se crearán estados (valores en sesiones) en la memoria de uno de los servidores del web-farm, pero si la siguiente petición http se redirecciona a otro servidor dentro del *clúster* (porque el balanceo sea 'sin afinidad'), entonces los valores obtenidos de la sesión ASP.NET serán incorrectos.

El modelo de proveedores de sesiones ASP.NET hasta ahora ofrecía tres proveedores, y ahora añadimos un cuarto:

- ***1.- Proveedor InProc***: En la memoria del servidor web, problema descrito anteriormente.

- ***2.- Proveedor StateServer***: Un servidor único con todas las sesiones en su memoria. Este proveedor es problemático, pues por un lado no es escalable, solo puede ser un único servidor ofreciendo el servicio de los valores de las sesiones. Adicionalmente, es un único punto de fallo. Si el servicio de este servidor de sesiones 'se cae', dejaran de funcionar todas las sesiones en nuestro web-farm de ASP.NET.

- ***3.- Proveedor SQLServer:*** Soluciona el problema de sesiones diferentes en un balanceo puro del web-farm y también soluciona el problema de tener un único punto de fallo (siempre y cuando dispongamos de un clúster SQL Server). Pero este proveedor tiene el inconveniente de disminuir el rendimiento e incluso la escalabilidad de nuestra aplicación, pues SQL Server tiene un techo de escalabilidad vertical.

- ***4.- Proveedor AppFabric-Cache:*** Este nuevo proveedor de sesiones ASP.NET utiliza el cache distribuido de AppFabric así como su alta disponibilidad, como un repositorio de sesiones ASP.NET. Este sistema se implementa de forma transparente y sin requerir ningún cambio en el código de las aplicaciones ASP.NET. Lo único necesario a cambiar es la definición XML del proveedor de sesión.

El proveedor de sesiones ASP.NET basado en el cache de AppFabric permite que las sesiones se mantengan vivas aun cuando alguno de los servidores del frontal ASP.NET se apague o 'se caiga', porque dichas sesiones están guardadas 'out'of-process' en el web-farm de cache de AppFabric.

Una vez se tiene instalado y configurado el cache de AppFabric, se debe crear un 'cache-nombrado' para almacenar las sesiones ASP.NET y posteriormente habilitar el proveedor ***DataCacheSessionStoreProvider*** modificando el Web.config, como se muestra en la figura.

```
Web.config

<?xml version="1.0"?>
<configuration>
  <configSections>
    <section name="dataCacheClient"
      type="Microsoft.Data.Caching.DataCacheClientSection,
CacheBaseLibrary"
      allowLocation="true" allowDefinition="Everywhere"/>
    <section name="fabric"
      type="System.Data.Fabric.Common.ConfigFile, FabricCommon"
      allowLocation="true" allowDefinition="Everywhere"/>
    <!-- AppFabric Cache -->
  </configSections>
  <dataCacheClient deployment="routing">
    <localCache isEnabled="false"/>
    <hosts>
      <!--List of services -->
      <host name="localhost" cachePort="22233"
        cacheHostName="DistributedCacheService"/>
    </hosts>
  </dataCacheClient>
  <fabric>
    <section name="logging" path="">
      <collection name="sinks" collectionType="list">
        <!--LOG SINK CONFIGURATION-->
        <!--defaultLevel values: -1=no tracing;
            0=Errors only;
            1=Warnings and Errors only;
            2=Information, Warnings and Errors;
            3=Verbose (all event information)-->
        <customType
          className="System.Data.Fabric.Common.EventLogger,FabricCommon"
          sinkName="System.Data.Fabric.Common.ConsoleSink,FabricCommon"
          sinkParam="" defaultLevel="-1"/>
        <customType
          className="System.Data.Fabric.Common.EventLogger,FabricCommon"

sinkName="System.Data.Fabric.Common.FileEventSink,FabricCommon"
          sinkParam="DcacheLog/dd-hh-mm" defaultLevel="-1"/>
        <customType
          className="System.Data.Fabric.Common.EventLogger,FabricCommon"
          sinkName="Microsoft.Data.Caching.ETWSink, CacheBaseLibrary"
          sinkParam="" defaultLevel="-1"/>
      </collection>
    </section>
  </fabric>
<appSettings/>
<connectionStrings/>
<system.web>
  <sessionState mode="Custom" customProvider="
AppFabricCacheSessionProvider">
    <providers>
      <add name="AppFabricCacheSessionProvider"
        type="Microsoft.Data.Caching.DataCacheSessionStoreProvider,
ClientLibrary"
        cacheName="session"/>
    </providers>
  </sessionState>
```

5.2.3.- Implementación de Cache en Nivel Cliente de Aplicaciones N-Tier (Rich-Client y RIA)

Una solución de tipo '*Rich-Client*' (también denominadas '*Smart-Client*', por ejemplo implementada con *WPF, VSTO o WinForms*) y también las aplicaciones RIA (implementadas por ejemplo con *Silverlight*) no están fuertemente acopladas con la capa de componentes de negocio ni la base de datos, gracias al 'consumo' de Servicios-Web, los cuales son 'débilmente acoplados', por diseño.

Precisamente, una de las principales ventajas de aplicaciones de tipo '*Smart-Client*' es el poder funcionar en un modo '*off-line/on-line*' simultáneamente. Este modo 'off-line' (trabajo desconectado en el PC cliente) favorece extremadamente el poder realizar también un ***caché*** en la capa de presentación '*Smart-Client*'.

El caché en aplicaciones distribuidas '*N-Tier*' que acceden a Servicios-Web y Servicios-WCF, es extremadamente importante, pues dicho ***caché*** puede reducir drásticamente la carga y número de peticiones a los Servicios-web, aumentando en gran medida el rendimiento global de la aplicación así como los tiempos de respuesta ofrecidos al cliente.

Los datos '*candidatos*' para ser cacheados en la capa cliente de una aplicación '*Rich-Client*' y *RIA* son todos aquellos datos que no cambien muy a menudo y sin embargo tengan que ver con todo el funcionamiento interrelacionado de los formularios. Por ejemplo, entidades de datos maestros de tipo '*Países'*, '*Provincias'*, etc. deberían de estar siempre cacheados en la memoria global del proceso cliente, cargándose al arrancar la aplicación o similar.

En caso de querer cachear datos que cambian más a menudo, es importante utilizar algún sistema que sea capaz de detectar cambios y/o refrescar el caché más a menudo (*time-outs* cortos).

En este caso (aplicación Rich ó RIA), el caché debe estar situado en algún punto global de la aplicación cliente (.exe), en el espacio de memoria del proceso principal.

Situando la gestión del caché de entidades de negocio en las clases ***Agentes de Servicios***, nos permite crear una capa 'inteligente' que en ciertos casos acceda on-line a los datos (acceso on-line mediante Servicios-Web/SOA) y en otros casos los obtenga del *caché* cliente local.

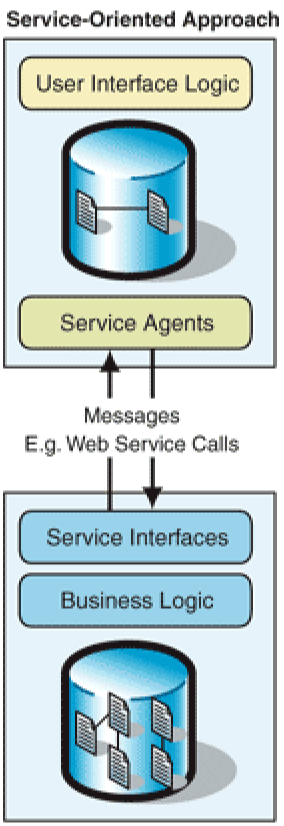


Figura 11.-

5.3.- Implementación de Logging/Registro

Existen librerías de carácter general, como la Enterprise Library de Microsoft Pattern & Practices que son muy útiles para implementar un logging complejo, con diferentes posibilidades.

Algunas implementaciones interesantes son:

- *Microsoft Enterprise Library Logging Building Block*
- *NLog*
- *Log4net*

5.4.- Implementación de Validación

Existen librerías de carácter general, como la Enterprise Library de Microsoft Pattern & Practices que son muy útiles para implementar un sistema reutilizable de VALIDACION de entrada de datos (*Microsoft Patterns & practices Enterprise Library Validation Block*).

CAPÍTULO 10

Arquetipos de Aplicación

Cuando construimos un sistema, lo más importante es identificar bien el tipo de aplicación que vamos a desarrollar. El tipo de aplicación que construiremos dependerá directamente de las restricciones de despliegue que tengamos y del servicio que tengamos que ofrecer. Por ejemplo, nuestro sistema puede estar limitado a no requerir ningún tipo de instalación en los clientes, en cuyo caso tendríamos una aplicación web clásica, o podría requerir una alta disponibilidad y acceso sin internet, en cuyo caso podría ser una aplicación móvil. En general, tenemos 5 tipos básicos de aplicaciones que engloban la mayor parte del espectro de las aplicaciones.

- **Aplicaciones de Escritorio** (*Rich Client Applications*): Nos ofrecen el mejor rendimiento y la mejor capacidad de aprovechar los recursos de la máquina. Pueden ser aplicaciones que usen internet o no. Tienen que instalarse en las máquinas cliente y son dependientes de la plataforma.

- **Aplicaciones web** (*Web Client Applications*): Únicamente necesitamos un navegador para acceder a ellas. El consumo de recursos en el cliente es muy bajo, se centra sobre todo en el servidor. No necesitan ser instaladas en el cliente y son multiplataforma.

- **Aplicaciones de servicios** (*Service Applications*): Son aplicaciones pensadas para ser utilizadas por otras aplicaciones. Las aplicaciones externas consumen los servicios que ofrece nuestra aplicación. Ofrecen una interfaz muy desacoplada. Se podría decir que una aplicación web es una aplicación de servicios con interfaz gráfica integrada.

- **Aplicaciones RIA** (*Rich Internet Applications*): Son aplicaciones que se ejecutan dentro del navegador. Ofrecen características similares a las aplicaciones de escritorio con la ventaja de que no es necesario instalarlas. Necesitan de un entorno de ejecución como Silverlight, Java FX o Flash.

- **Aplicaciones móviles** *(Mobile applications*): Son aplicaciones diseñadas para ejecutarse sobre teléfonos móviles, PDAs, etc. Lo peor que tienen es la limitación de recursos de la máquina y el pequeño tamaño de la pantalla, por otra parte ofrecen una alta disponibilidad y facilidad de uso, así como la capacidad de soportar trabajar conectados o desconectados.

La principal decisión a tomar es cuando utilizar un tipo de aplicación u otro, para ello aquí se exponen una serie de consejos.

- **Aplicaciones ricas de Escritorio**
    - La aplicación tiene que soportar trabajar tanto conectada como desconectada de la red.
    - La aplicación se instalará en los PCs de los clientes.
    - La aplicación tiene que ser altamente interactiva y dar una buena respuesta.
    - La aplicación tiene que ofrecer mucha funcionalidad pero no necesita grandes gráficos o multimedia.
    - La aplicación tiene que usar recursos del PC.
- **Aplicaciones Web**
    - La aplicación no requiere de una gran interfaz gráfica ni el uso de recursos multimedia.
    - Se busca la simplicidad del despliegue web.
    - La aplicación debe ser multiplataforma.
    - La aplicación tiene que estar disponible por internet.
    - La aplicación debe minimizar el uso de recursos y las dependencias en el cliente.
- **Servicios**
    - La aplicación expondrá una funcionalidad que no requiere de interfaz gráfica.
    - La aplicación debe estar muy desacoplada de los clientes.
    - La aplicación debe ser compartida o consumida por otras aplicaciones.
    - La funcionalidad de la aplicación debe ser consumida a través de la red.

1.- ARQUETIPO 'APLICACIÓN WEB'

Las aplicaciones web se adecúan a un estilo arquitectural del tipo Cliente / Servidor. En este paradigma el cliente (Navegador) consume y visualiza los servicios ofrecidos por el servidor (Páginas web). La lógica de una aplicación web se suele concentrar en el servidor. Las formas normales de organizar una aplicación web son N-Capas y MVC. Para las dos aproximaciones tenemos ASP.NET y ASP.NET MVC.

A la hora de diseñar la arquitectura de una página web es muy importante entender la naturaleza "sin estado" del protocolo HTTP, pues esto condiciona en gran medida el diseño. Cuando hablamos de aplicaciones en ASP.NET el enfoque clásico es la estructuración de la aplicación en N-Capas, el cual puede estar particularizado hacia tendencias DDD según se explica en esta guía de Arquitectura:

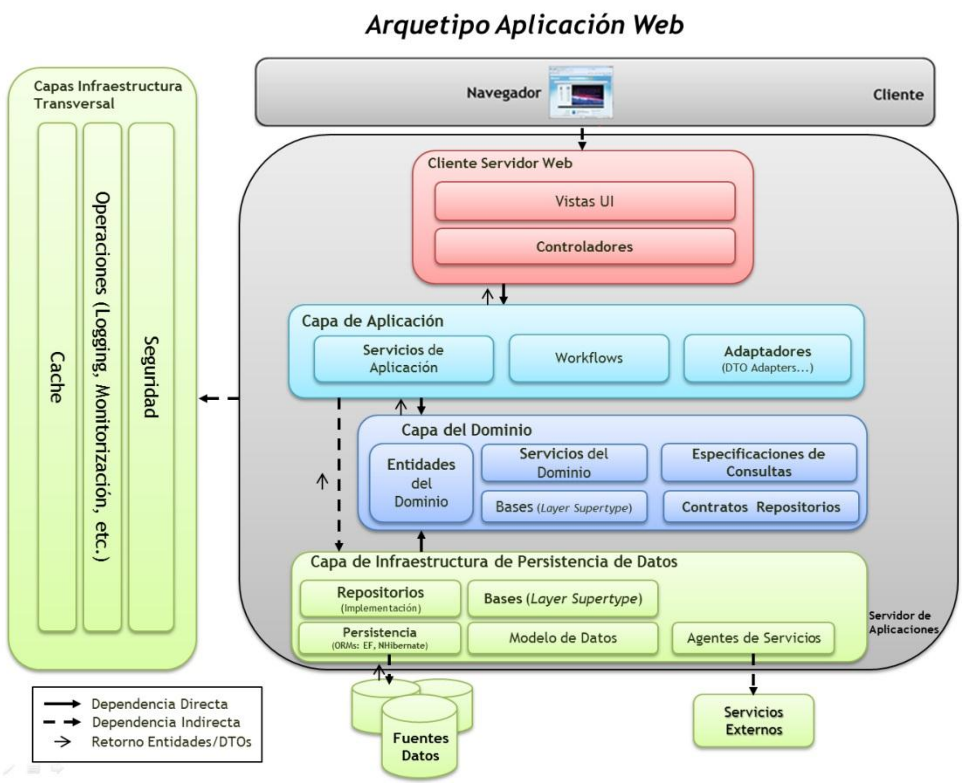


Figura 1.- Arquetipo Aplicación Web

A la hora de diseñar aplicaciones web es muy importante tener en cuenta una serie de aspectos en las fases iniciales de diseño de la arquitectura, independientemente del estilo arquitectural seleccionado. Las principales recomendaciones son:

- **Particiona la aplicación de forma lógica**: Separar el diseño en distintas capas permite obtener un código más reutilizable, más escalable y más testeable.

- **Minimiza el acoplamiento entre capas**: Esto puede conseguirse usando clases base e interfaces (preferiblemente) que sirvan como abstracción para la comunicación entre capas. La idea es diseñar componentes con un mínimo conjunto de operaciones con unos parámetros y valores de retorno bien conocidos. Es importante que las operaciones sean las mínimas y que presenten acciones atómicas más que un diálogo entre las capas.

- **Conoce como los componentes se comunican entre sí:** Es importante tener en cuenta si la comunicación entre capas se realiza a través de medios físicos (Red) o todos los componentes se ejecutan en el mismo proceso.

- **Minimiza las comunicaciones:** Cuando diseñamos una aplicación web es importante minimizar las comunicaciones entre cliente y servidor. Las comunicaciones deben limitarse a las estrictamente necesarias. Para ello se pueden usar técnicas de cacheo de datos en el navegador y de cacheo de páginas en el servidor.

- **Caching:** Una buena estrategia de caching es lo que diferencia a una mala aplicación de una buena aplicación. El cacheo mejora la eficiencia y la escalabilidad de la aplicación y reduce el tiempo de respuesta de las peticiones.

- **Guarda registros de la actividad de la aplicación**: Los registros de actividad permiten detectar y analizar post-mortem los funcionamientos anómalos de una aplicación en producción, así como detectar las actividades sospechosas con el objetivo de prevenir ataques.

- **Evita los bloqueos en operaciones largas**: Si la aplicación debe realizar una operación costosa lo mejor es utilizar un procesamiento asíncrono que permita al servidor realizar otras tareas.

- **Autentifica a los usuarios en cada entrada de una zona de confianza**: Autentificar a los usuarios a la entrada de cada zona de confianza permite separar físicamente los distintos niveles y minimizar los riesgos de que un usuario adquiera privilegios que no tiene.

- **No envíes información sensible sin codificar:** Siempre que haya que enviar información sensible como un usuario y un password se debe usar SSL o bien cifrar y signar el contenido para garantizar la confidencialidad y la autoría de la información enviada.

- **Minimiza los privilegios de la aplicación**: Es conveniente que la aplicación se ejecute con los menores privilegios posibles ya que si un atacante toma el control de la misma tendrá menos capacidad de causar daños.

2.- ARQUETIPO 'APLICACIONES RIA'

Las aplicaciones RIA (*Rich Internet Application*) son la elección más adecuada cuando queremos dar una experiencia de usuario más visual y con mejor respuesta a través de la red. Las aplicaciones RIA ofrecen la calidad gráfica de una aplicación de escritorio y las ventajas de despliegue y mantenimiento de una página web. Las aplicaciones RIA siguen un estilo arquitectural Cliente / Servidor al igual que las páginas web, no obstante, en este tipo de aplicaciones parte de la lógica de negocio se pasa al cliente para mejorar la respuesta y reducir la carga en el servidor. De la misma forma, las RIAs se programan en lenguajes más potentes como C# o VB.Net en lugar de con AJAX como las aplicaciones web lo cual permite realizar desarrollos mucho más complejos. Además las RIAs son realmente multiplataforma y multinavegador ya que se ejecutan gracias a un plug-in en el navegador. La estructura general de una aplicación ría es un modelo en N-Capas como se puede ver a continuación:

Arquetipo Aplicación RIA

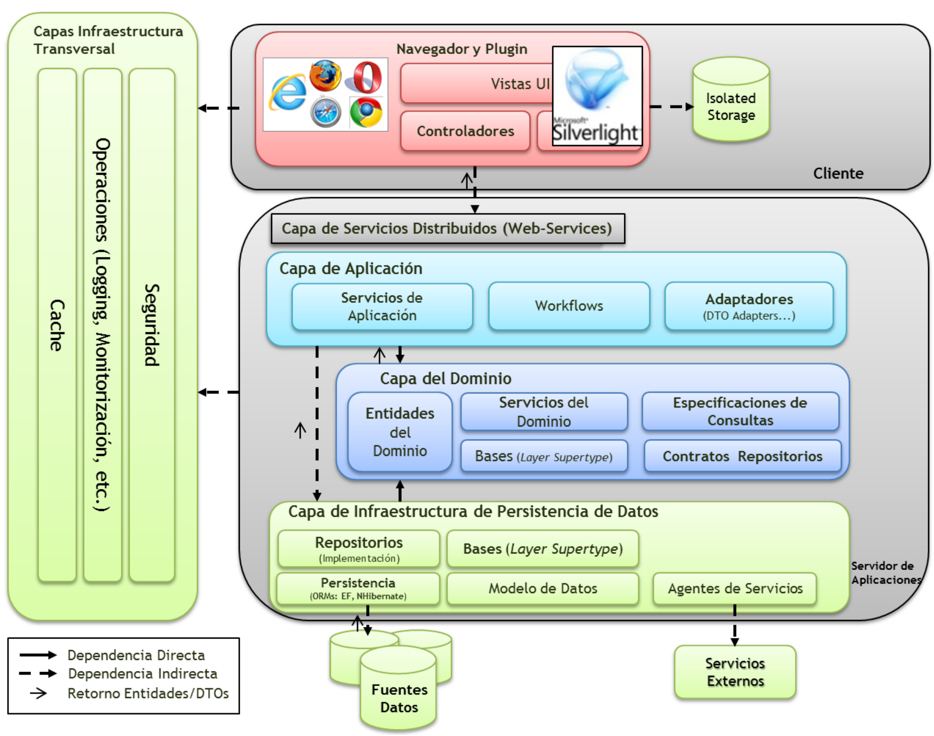


Figura 2.- Arquetipo Aplicación RIA

Debemos entender que las tecnologías RIAs como Silverlight se ejecutan en un subconjunto reducido de la plataforma de desarrollo general. Por ello, lo normal es que la lógica de la aplicación se concentre en el servidor y que la parte del cliente sea solo

la interfaz gráfica. La comunicación entre el cliente y el servidor se realiza estrictamente a través de servicios web usando HTTP.

La estructura general de las capas de una aplicación RIA sueles ser en N-Capas, como la explicada en la presente guía de Arquitectura N-Capas donde además incluimos tendencias DDD.

En el caso de una aplicación RIA, es completamente necesario hacer uso de la Capa de Servicios Distribuidos, pues debemos comunicarnos con la lógica de negocio y acceso a datos del servidor de una forma remota y desde la programación cliente:

- **Capa de servicios distribuidos:** Es imprescindible en una aplicación RIA. Se encarga de permitir el acceso a los datos y a la lógica de negocio implementada en el servidor.

A la hora de decidir si el sistema que tenemos pensado desarrollar encaja bien dentro del tipo de aplicación RIA podemos apoyarnos en las siguientes consideraciones de diseño:

- **Escoge una RIA en base a los usuarios, la calidad de la interfaz y la facilidad de despliegue:** Considera desarrollar una RIA cuando los potenciales usuarios tengan un sistema operativo y un navegador compatibles. Si no es así, considera si la aplicación perderá muchos potenciales usuarios por este motivo. Las aplicaciones RIA ofrecen la misma facilidad de despliegue y más capacidades gráficas que las aplicaciones web, además son más independientes del navegador y soportan mejor los escenarios de streaming.

- **Diseña la aplicación para usar servicios:** La forma normal en que una RIA debería trabajar es consumiendo servicios web para realizar la lógica de negocio de la aplicación. De esta forma se facilita la reutilización de la infraestructura web existente. Solo se debería transferir lógica de negocio al cliente por razones de eficiencia o para mejorar la respuesta de la interfaz gráfica.

- **Ten en cuenta las restricciones de ejecutarse dentro del navegador:** Las RIAs se ejecutan dentro del navegador, por lo que no tienen acceso o tienen acceso limitado a los recursos del usuario como pueden ser el sistema de ficheros, cámaras, etc.

- **Determina la complejidad de la interfaz:** Las RIAs son más adecuadas cuando todo se puede hacer desde una pantalla. Las interfaces con varias pantallas requieren más trabajo para implementar el flujo de acciones. Es muy importante modificar el historial y los botones de página anterior y página siguiente para ofrecer un funcionamiento coherente.

- **Identifica los principales escenarios para mejorar la eficiencia y la respuesta de la aplicación:** Examina los escenarios para identificar como dividir y cargar los módulos. Es muy importante realizar una carga bajo demanda de los módulos para reducir el tiempo de descarga de la aplicación y

trasladar la lógica de negocio costosa computacionalmente al cliente para mejorar el tiempo de respuesta disminuyendo la carga en el servidor.

- **Contempla la posibilidad de que los usuarios no tengan el plug-in RIA instalado:** Ofrece una alternativa para los usuarios que no tengan instalado el plugin RIA. Considera si los usuarios tendrán permiso y querrán instalar el plug-in y ofrece un acceso directo a la descarga o una interfaz web alternativa.

3.- ARQUETIPO 'APLICACIÓN RICA DE ESCRITORIO' (RICH CLIENT)

Las aplicaciones de escritorio son el tipo de aplicación tradicional. Nos ofrecen potentes interfaces gráficas y alto rendimiento. Pueden funcionar en todo tipo de entornos; con conexión, sin conexión o con conectividad ocasional. Este tipo de aplicaciones puede ejecutarse solo en una máquina cliente o conectarse con servicios de ofrecidos por un servidor. Las aplicaciones de escritorio ofrecen la mayor flexibilidad en cuanto a recursos, topologías y tecnologías a usar. Por norma general las aplicaciones cliente suelen estructurarse en N-Capas, opcionalmente con tendencias DDD, como proponemos en nuestra Arquitectura:

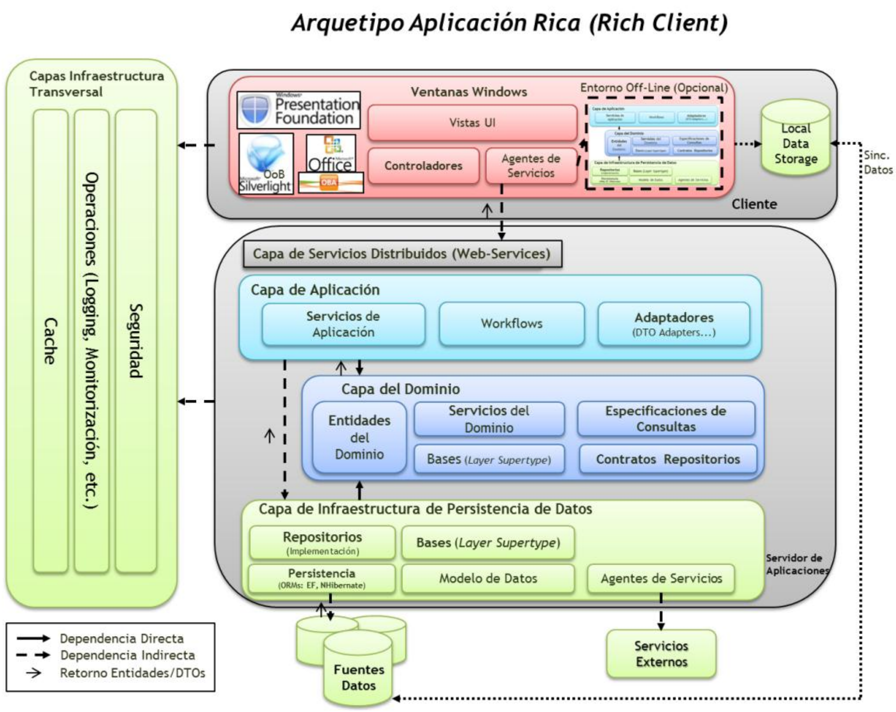


Figura 3.- Arquetipo Aplicación Rica

La estructura de capas es similar a la arquitectura en una aplicación RIA, sin embargo en una aplicación Rica también se contemplan casos más complejos de escenarios Off-Line, donde se replican ciertas partes de capas típicas del servidor también en el cliente para poder trabajar en modo off-line.

Las aplicaciones de cliente ofrecen mucha flexibilidad por lo que es normal encontrar estilos arquitecturales muy diversos. Desde aplicaciones complejas que almacenan su estado en la máquina del cliente hasta aplicaciones que se conectan a servicios web para obtener los datos y la lógica que necesitan.

Por este motivo, la arquitectura de las aplicaciones de escritorio puede variar bastante dependiendo de la aplicación de escritorio específica. En general, todas seguirán un estilo arquitectural de N-Capas, lo que cambiará será el número de capas y la localización física de las mismas.

Componentes clave

El éxito de una aplicación de escritorio depende en gran medida de la capacidad del arquitecto para escoger la tecnología adecuada y diseñar una estructura que minimice el acoplamiento entre los componentes y maximice la cohesión de los mismos. Para ello podemos seguir las siguientes directrices:

- **Escoge la tecnología apropiada basada en los requisitos de la aplicación:** En el entorno de .NET Framework tenemos disponible Windows Forms, Windows Presentation Foundation y Office Business Applications.

- **Separa el proceso de la interfaz de la implementación de la misma:** Utilizar patrones de diseño como Model-View-Controller o Model-View-View-Model mejora el mantenimiento, la reusabilidad y el testeo de la aplicación.

- **Identifica las tareas que debe realizar el usuario y los flujos de ejecución de dichas tareas:** De esta forma es más sencillo diseñar las pantallas y los pasos a realizar para conseguir un objetivo.

- **Extrae todas las reglas de negocio y tareas no relacionadas con la interfaz:** La interfaz gráfica únicamente debería mostrar los datos de la aplicación e indicar a la capa inmediatamente inferior qué acción ha llevado a cabo el usuario sobre ella misma.

- **Desacopla tu aplicación de cualquier servicio web que use:** Emplea para ello una interfaz basada en mensajes para comunicarte con cualquier servicio en otro nivel físico distinto.

- **Evita el acoplamiento con objetos de otras capas:** Usa extensivamente interfaces, clases base o mensajes entre capas para separar las abstracciones de las implementaciones.

- **Reduce las comunicaciones con capas remotas:** Crea métodos que se ejecuten asíncronamente y que permitan realizar una acción completa. Es prioritario evitar los diálogos entre niveles físicos distintos.

4.- ARQUETIPO SERVICIO DISTRIBUIDO - SOA

Los Servicios Distribuidos son sistemas que están pensados para que otros programas los utilicen remotamente. Este tipo de sistemas ofrece un conjunto de funcionalidad a otras aplicaciones con un nivel de acoplamiento muy bajo. Para ello emplean una interfaz basada en mensajes, de esta forma las características de la comunicación pueden ser acordadas en tiempo de ejecución (tipo de canal, cifrado, autentificación, formato de los mensajes…) lo que permite un desacoplamiento total entre el sistema que oferta los servicios y las aplicaciones que los consumen.

Podemos entender cada servicio ofertado por este tipo de sistemas como una llamada remota a un método que efectúa algún cambio en el estado en el servidor o que devuelve un resultado. Veamos las distintas capas de una aplicación de servicios en más detalle:

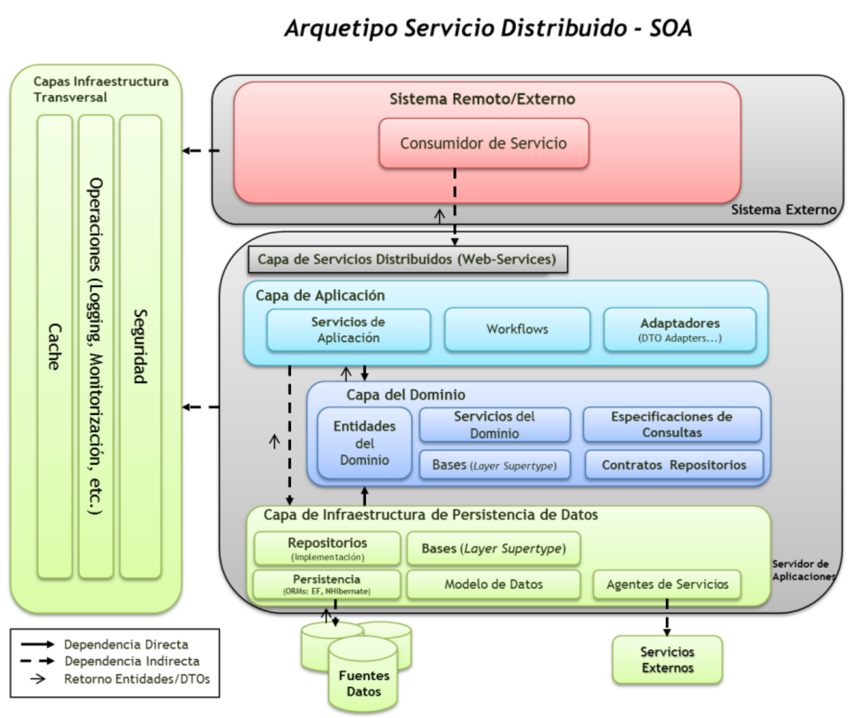


Figura 4.- Arquetipo Servicio Distribuido (SOA)

Como podemos observar, en una aplicación de servicios (SOA) las capas internas son muy similares a las de otro tipo de aplicaciones. Podría pensarse que la capa de servicios también es idéntica a la capa de otro tipo de aplicaciones, pero esto no es así. Las capas de servicios de una aplicación N-Tier está pensada, en general, para que esos servicios sean consumidos internamente por las capas del cliente de la aplicación y solo por este, en ese caso tendríamos un control del desarrollo extremo a extremo. En este entorno, los arquitectos conocen bastante bien la topología de despliegue de la aplicación y otros factores a tener en cuenta de forma que pueden obviar ciertas consideraciones y recomendaciones por no ser necesarias.

En las aplicaciones de servicios SOA esto no es así. El arquitecto tiene que planear muy cuidadosamente el conjunto de servicios a ofertar, de forma que sean mínimos, oferten la funcionalidad completa y **ofrezcan al consumidor contratos de datos (DTOs) desacoplados de los objetos internos del servicio (Entidades del Dominio). Este punto es fundamental para que las aplicaciones consumidoras puedan evolucionar a diferente ritmo que el Servicio SOA**. Está relacionado precisamente con uno de los principios de SOA: '***Los Servicios en SOA deben ser Autónomos***'.

Además de la especial atención en el diseño de los servicios, es muy importante tener en cuenta todos los aspectos relativos a seguridad, formatos, entornos, etc. del servicio.

En general, un servicio puede ser desplegado a través de internet, en una intranet, en una máquina local o en cualquier combinación de las tres posibilidades. Los servicios desplegados a través de internet requieren decisiones sobre autorización, autenticación, límites de zonas seguras etc. Usando para ello claves, certificados, etc. Los servicios a consumir por una intranet, pueden basar su seguridad en sistemas como *Active Directory* mientras que los servicios que se consumen en la misma máquina pueden no requerir protección dependiendo de los usuarios y seguridad de la misma. Los servicios que se despliegan en varios de estos entornos a la vez deberían implementar mecanismos de autentificación y seguridad acordes a cada una de las posibilidades.

Ya hemos hablado sobre lo que es una aplicación de servicios y sobre los entornos en los que se puede desplegar. No obstante, independientemente del entorno de despliegue existen una serie de consideraciones comunes a tener en cuenta a la hora de diseñar nuestra aplicación de servicios:

- **Diseña operaciones generales:** A la hora de trabajar con servicios, uno de los factores que más limita la escalabilidad de los mismos es la carga de aceptar y responder peticiones. Entre otras cosas, los servicios crean un nuevo thread para atender a cada petición entrante, por tanto si se hacen servicios muy pequeños la cantidad de procesamiento realizado no compensa el coste de aceptar y responder la petición. Esto unido a que se necesitan más llamadas para realizar una misma acción es lo que provoca la pérdida de escalabilidad. Por tanto, en la medida de lo necesario los servicios deberían seguir un esquema de Mensaje-Respuesta más que una comunicación "dialogada".

- **Diseña entidades extensibles:** Los servicios como todo el software evolucionan a lo largo del tiempo, debido a esto podemos tener que extender

nuestras entidades en un futuro. Estas extensiones deberían realizarse sin afectar a los clientes actuales del servicio.

- **Compón las entidades de servicio con tipos básicos:** Las entidades de servicio deberían utilizar tipos básicos como int, float, string, etc. antes que usar tipos definidos ya que esto mejora la flexibilidad y compatibilidad del servicio.
- **Diseña solo para el contrato del servicio:** Los servicios deben ofrecerse a los consumidores a través de una interfaz y nunca de su implementación. De la misma forma, no se debe implementar ninguna funcionalidad que no esté reflejada en la interfaz.
- **Diseña el servicio asumiendo la posible llegada de peticiones incorrectas:** Nunca se debe asumir que los mensajes recibidos son correctos. Al igual que con cualquier otra fuente externa, debe realizarse una validación de los datos.
- **Separa las responsabilidades de la lógica de negocio de las responsabilidades de la infraestructura de servicios:** La lógica de negocio debe tener una implementación independiente de la capa de servicios. Con esto no solo nos referimos a las dependencias entre componentes de las mismas, sino a su estructura en sí. La capa de negocio no debería diseñar sus operaciones teniendo en cuenta que luego van a ser desplegadas como servicios. Hacer el mapeo servicio/negocio es una de las responsabilidades de la capa de servicio.
- **Asegúrate de que el servicio soporta la llegada de mensajes repetidos:** El servicio debe poder aguantar que le llegue un mismo mensaje más de una vez y tratar esta casuística de forma coherente. Un mensaje puede llegar repetido porque un origen lo haya enviado varias veces o porque varios orígenes hayan enviado el mismo mensaje.
- **Asegúrate de que el servicio soporta la llegada de mensajes desordenados:** El servicio debe poder soportar que le lleguen mensajes desordenados, almacenándolos y procesándolos posteriormente en el orden correcto.
- **Diseña servicios orientados a la aplicación y no a componentes específicos:** Los servicios deberían implementar operaciones del ámbito de la aplicación más que realizar colaboraciones con otros componentes del cliente para realizar una tarea. Si por ejemplo tuviésemos una aplicación de gestión de cuentas bancarias, el servicio tendría operaciones tales como "RealizarTransferencia(CuentaOrigen,CuentaDestino)". Una transferencia nunca se implementaría como "SacarDinero(Origen)" -> MeterDinero(Destino).
- **Desacopla la interfaz del servicio de su implementación:** En una aplicación de servicios no deben exponerse nunca las entidades de negocio. Los servicios deben basarse en contratos con los que los clientes interactúan y es la capa de servicios la responsable de traducir dichos contratos a la capa de negocios.

- **Las aplicaciones de servicios deben tener límites muy claros:** El acceso a la funcionalidad de la aplicación debe realizarse única y exclusivamente a través de los servicios que esta oferta.
- **Los servicios deben ser autónomos:** Los servicios no deben requerir ni asumir nada acerca del cliente de los mismos por lo que deben protegerse contra impersonaciones y peticiones malformadas.
- **La compatibilidad del servicio debe estar basada en políticas:** Los servicios deben publicar un documento de políticas que defina como los clientes pueden interactuar con el servicio.

5.- ARQUETIPO APLICACIONES MÓVILES

Construir una aplicación para un dispositivo móvil tal como una PDA o un teléfono puede convertirse en todo un reto para el arquitecto. Esto es así debido a los numerosos factores limitantes a tener en cuenta como el tamaño de la pantalla o de la memoria y las capacidades limitadas de almacenamiento, procesamiento y conexión.

Las aplicaciones para móviles siguen también una arquitectura en N-Capas, con una arquitectura muy similar a una aplicación Rica de escritorio, pero donde la tecnología de presentación está limitada al entorno móvil, tecnológicamente más limitado:

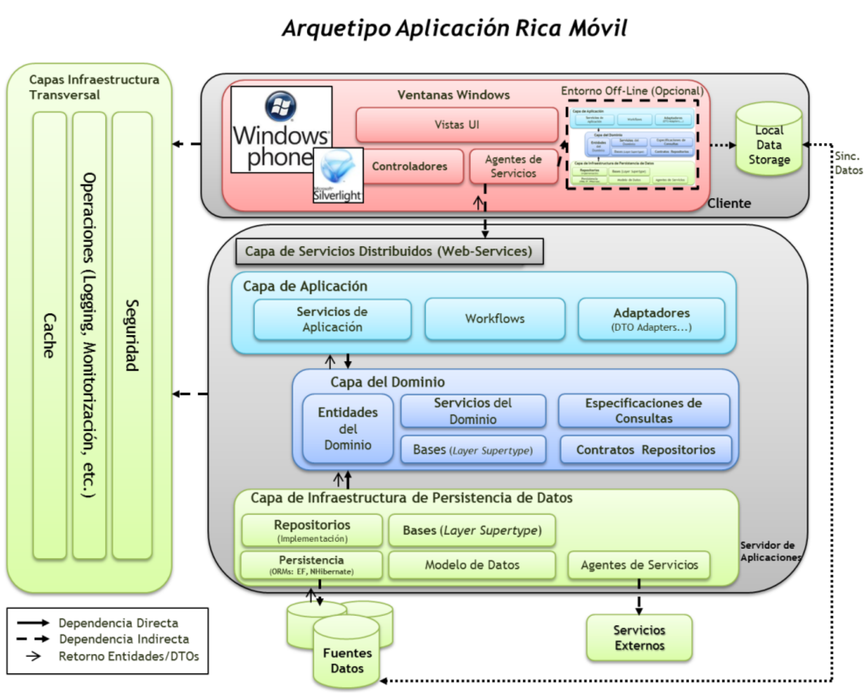


Figura 5.- Arquitectura Arquetipo Aplicación Móvil

- **Capa de presentación:** La capa de presentación es minimalista dado el pequeño tamaño de las pantallas de los dispositivos. Las interfaces son todas mono-ventana y los diálogos dirigen el flujo de la aplicación. Es mejor usar botones y otros componentes que funcionan mediante selección frente a cajas de texto dada la limitación para introducir texto de estos dispositivos. Así mismo los controles se hacen más grandes para permitir una mejor experiencia táctil.

Las aplicaciones móviles tienen muchas restricciones que limitan lo que podemos hacer con ellas e indican lo que no debemos hacer. Lo normal es que una aplicación móvil siga estas recomendaciones:

- **Decide el tipo de aplicación a construir:** Si la aplicación puede trabajar desconectada o con conexión intermitente lo mejor es una aplicación de cliente. Si la aplicación depende de un servidor al que está conectada es mejor una aplicación web.

- **Determina el tipo de dispositivos que la aplicación soportará**: A la hora de desarrollar una aplicación móvil hay que plantearse el tamaño de la pantalla de los dispositivos, su resolución, memoria, CPU, almacenamiento y hardware disponible.

- **Diseña la aplicación considerando escenarios de conexión intermitente y limitada:** La aplicación no siempre va a tener conexión, por lo que es importante diseñar estrategias de cacheo de datos, mantenimiento del estado y sincronización de la misma.

- **Diseña la interfaz teniendo en cuenta las restricciones de los dispositivos:** Como ya hemos dicho los dispositivos móviles tienen una pantalla muy pequeña por lo que las interfaces gráficas deben ser muy simples. Además para el usuario es más fácil seleccionar elementos que escribir texto, por lo que siempre que se pueda es preferible la primera opción. Este tipo de interfaces suele ser táctil, por lo que los elementos seleccionables deben tener un tamaño suficiente para pulsarlos fácilmente con los dedos.

- **Diseña una arquitectura por capas adaptada a los dispositivos móviles:** Este tipo de aplicaciones suele agrupar varios niveles dentro del dispositivo móvil. Por eso es mejor realizar diseños más simples que los de las aplicaciones tradicionales que reduzcan la memoria usada y mejoren el rendimiento de la aplicación.

- **Diseña la aplicación considerando las limitaciones de los dispositivos:** Cada decisión de diseño de la aplicación debe tener en cuenta las limitaciones de CPU, memoria, almacenamiento y batería de los dispositivos. Por ello es importante apagar los periféricos que no estén en uso y usar herramientas

como SQL Server Compact Edition para evitar que por falta de memoria el sistema operativo borre datos de nuestra aplicación.

6.- ARQUETIPO 'APLICACIONES CLOUD COMPUTING '

Las aplicaciones en la nube son un nuevo fenómeno en la industria de las aplicaciones, todo el mundo habla de la nube, pero realmente no existe una definición universalmente aceptada sobre lo que es la nube y el *Cloud Computing*.

En cualquier caso, ***Cloud Computing* a nivel de plataforma significa un entorno concreto de despliegue de aplicaciones, por lo que la Arquitectura lógica será muy similar a una aplicación desplegada en un datacenter propio, es decir, podríamos seguir también el esquema de una '*Arquitectura N-Capas Orientada al Dominio*' si la aplicación es compleja, o una arquitectura simplificada y RAD si la aplicación es sencilla.**

No es nuestra intención en este documento dar una definición precisa de lo que es Cloud Computing, pero sí daremos una idea básica sobre lo que es. ***Cloud Computing* es la evolución de SaaS y plataformas de *hosting* hacia un entorno de despliegue más dinámico y auto-servicio: La nube.** Software as a Service es una idea que ya lleva un tiempo con nosotros y que consiste simplemente en ofrecer un software como un servicio que se utiliza a través de la red. El ejemplo más claro es el correo electrónico como Hotmail. *Utility Computing* es la capacidad de ofrecer recursos computacionales (CPU, Almacenamiento y Red) de forma transparente al usuario que los consume bajo un modelo de pago por uso como si de agua, luz o gas se tratase.

Estos son los dos conceptos básicos que nos permitirán entender lo que es la nube. El siguiente paso es hacer una distinción entre los proveedores de Cloud Computing y los usuarios de Cloud Computing. Los primeros son importantes empresas con grandes datacenters que ofrecen los recursos de los mismos a través de Utility Computing. Generalmente ofrecen una plataforma para el alojamiento de sistemas o aplicaciones como Windows Azure. Los segundos, los Cloud Users son usuarios o empresas que desarrollan su aplicación para ejecutarla sobre la plataforma de los Cloud Providers. Los cloud users hacen esto por las numerosas ventajas que les ofrece la nube, como son:

- Gestión automática de la infraestructura hardware.
- Traslado de los costes capitales a costes operacionales (CAPEX -> OPEX).
- Pago por consumo.
- Reducción de costes.
- Escalabilidad.
- Consumo ajustado a la demanda.
- Alta disponibilidad (Infraestructura virtualizada).

Por otro lado las aplicaciones en la nube tienen también una serie de inconvenientes a considerar antes de mudar una aplicación entera:

- Las capacidades de configuración son limitadas.
- La conexión a la red es imprescindible.
- La confidencialidad de la información no está garantizada.
- No existe interoperabilidad directa entre distintos Cloud Providers.
- La auditoría es difícil de realizar.
- No existen contadores de rendimiento con acceso on-line para analizar el sistema.

Ya hemos definido qué es la nube y enumerado sus principales virtudes y defectos, a continuación veremos cuál es la estructura general de los Cloud Providers ya que a diferencia del resto de aplicaciones, en las aplicaciones en la nube tenemos una serie de niveles ya definidos en los que encajamos las capas de nuestra aplicación.

Por lo general los Cloud Providers ofrecen dos servicios en la nube: computación y almacenamiento. Las plataformas tipo Windows Azure buscan proveer a los usuarios de una infraestructura altamente escalable para el alojamiento de sus aplicaciones. Por esta razón, ofrecen un nivel de servicio sin conservación de estado y un nivel de almacenamiento con conservación de estado.

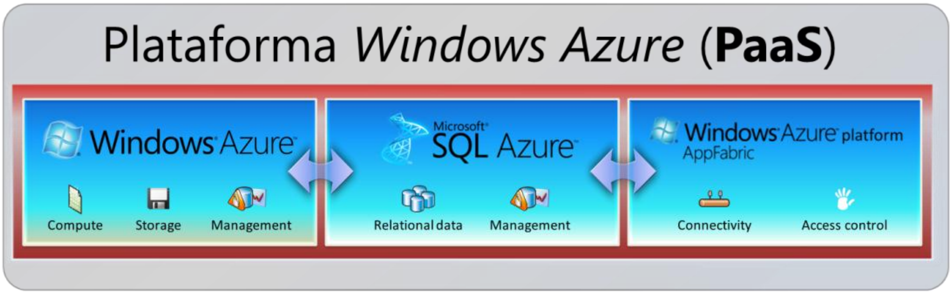


Figura 6.- Plataforma Servicios Windows Azure

- **Nivel de servicio:** El nivel de servicio aloja las aplicaciones de los usuarios. Dentro de este nivel se pueden alojar servicios web, aplicaciones web o aplicaciones de procesamiento de información en segundo plano (batch processing).
- **Nivel de almacenamiento:** El nivel de servicio aloja la información de las aplicaciones, estén estas en la nube o no. Expone una interfaz REST a través

de HTTP, por lo que cualquier aplicación puede hacer uso de él. Permite el almacenamiento de datos estructurados, BLOBs y mensajes para la sincronización.

Una aplicación puede estar completamente alojada en la nube y sus capas se mapearán completamente sobre los dos niveles definidos o puede usar solo uno de ellos. Además, los cloud providers suelen complementar su oferta con una amplia gama de servicios web que las aplicaciones pueden emplear como .NET Service Bus o .NET Access Control Services, SQL Azure o Live Services.

En esencia, el cloud computing nos ofrece una forma de abstraer la infraestructura física de nuestro sistema y nos garantiza una serie de ventajas como alta disponibilidad, escalabilidad bajo demanda y persistencia. Las aplicaciones en la nube no son un tipo especial de aplicación, sino que son especializaciones de tipos de aplicaciones conocidas a la infraestructura del proveedor de cloud. En esencia, los tipos de aplicación más adecuados para la nube son:

- **Aplicaciones móviles:** Este tipo de aplicaciones se puede beneficiar de los servicios de computación para solventar su falta de potencia y de los servicios de almacenamiento para poder permanecer en sincronía con otros dispositivos y disponer de copias de seguridad de los datos.

- **Aplicaciones web:** Estas aplicaciones son ideales para la nube ya que ofrecen una interfaz estándar a través de HTML y HTTP y garantizan que el sistema es multiplataforma.

- **Aplicaciones orientadas a servicios:** Este tipo de aplicaciones se ve beneficiada por la nube ya que resuelve todos los problemas de disponibilidad, orquestación, etc. que hay que afrontar cuando desarrollamos una aplicación de este tipo.

- **Aplicaciones Híbridas:** El resto de aplicaciones puede hacer uso de la infraestructura de la nube, bien de almacenamiento o bien de computación o disponer de funcionalidad ampliada cuando tenga conexión con la nube.

La infraestructura en niveles de los cloud providers nos influencia a la hora de elegir los estilos arquitecturales para nuestra aplicación. Lo normal en las aplicaciones en la nube es hacer un desarrollo en N-Capas con una capa de acceso a datos y una capa de negocio fijas y una capa de servicio opcional para los casos en los que sea necesario como aplicaciones RIA, híbridas u orientadas a servicios.

- **Capa de datos:** Esta capa se encarga de acceder a los datos. La información de nuestra aplicación debería estar organizada en información estructurada, BLOBs y mensajes. La información normal y de pequeño tamaño la almacenaremos en almacenamiento estructurado. Si alguna entidad de nuestro dominio tiene algún BLOB la carga debe ser diferida y nuestro DAL debe ofrecer métodos para ello. Por último, dado el carácter distribuido de las

aplicaciones en la nube, se debe usar la capa de datos para almacenar y recuperar toda la información de sincronización del sistema.

- **Capa de lógica del Dominio y Aplicación:** Estas capas se encargan de implementar la lógica de la aplicación y no debería diferenciarse en nada de ninguna otra capa de aplicación. Es importante que sea una capa sin estado ya que las aplicaciones en la nube funcionan mediante peticiones/respuestas y no queremos introducir ninguna afinidad con ningún servidor entre peticiones.

- **Capa de servicios distribuidos:** Esta capa es opcional y solo necesaria cuando se realizan aplicaciones orientadas a servicios o RIAs. Expone una interfaz de servicio para la funcionalidad de la aplicación. Es muy importante que proporcione operaciones de nivel de aplicación para minimizar la sobrecarga de las llamadas y maximizar el rendimiento.

- **Capa de presentación:** Esta capa expone una interfaz para la aplicación en la nube. La interfaz puede ser o bien RIA o bien Web. En cualquier caso funciona mediante un esquema de petición/respuesta por lo que es importante no almacenar ninguna información de estado en sesiones ya que esto introduce afinidad con el servidor, dificulta el balanceo de carga y limita en gran medida la escalabilidad.

Las aplicaciones en la nube tienen la ventaja de ser altamente escalables, pero la plataforma en sí no garantiza esto, sino que es responsabilidad del arquitecto seguir unas pautas que permitan la escalabilidad de la aplicación. A continuación presentamos las principales recomendaciones:

- Almacena el estado de la aplicación en el nivel de almacenamiento.

- No almacenes estado en el nivel de servicio.

- Utiliza colas para sincronizar las distintas instancias de los niveles de servicio.

- Almacena los elementos de gran tamaño en el almacenamiento BLOB.

- Utiliza el almacenamiento estructurado para la persistencia de entidades.

- Implementa mecanismos de concurrencia optimista en el almacenamiento estructurado para la gestión de la concurrencia si es necesario.

- Minimiza el tamaño y la cantidad de las comunicaciones.

- Divide en varios envíos los envíos de datos de gran tamaño.

7.- ARQUETIPO APLICACIONES OBA (OFFICE BUSINESS APPLICATIONS)

Las *Office Business Applications*, en adelante OBAs, son un tipo de sistemas empresariales compuestos. Realmente puede considerarse un sub-conjunto de las aplicaciones Cliente-Ricas (*Rich-Client*)

Las OBAs proporcionan la funcionalidad de cualquier aplicación de negocio clásica integrada dentro de la suite **Microsoft Office**. Es decir, las OBAs se componen de un conjunto de aplicaciones y servicios que solucionan un problema de negocio.

Las OBAs integran sistemas nuevos y viejos de Line-Of-Business (en adelante LOB). Las OBAs delegan la interfaz de usuario y la automatización de tareas en Office para simplificar las tareas complejas que requieren interacción humana.

Las OBAs siguen un estilo arquitectural en N-Capas aunque estas difieren un poco de las de una aplicación común dado que las OBAs integran aplicaciones existentes y no las construyen. El diagrama de la arquitectura se puede ver a continuación:

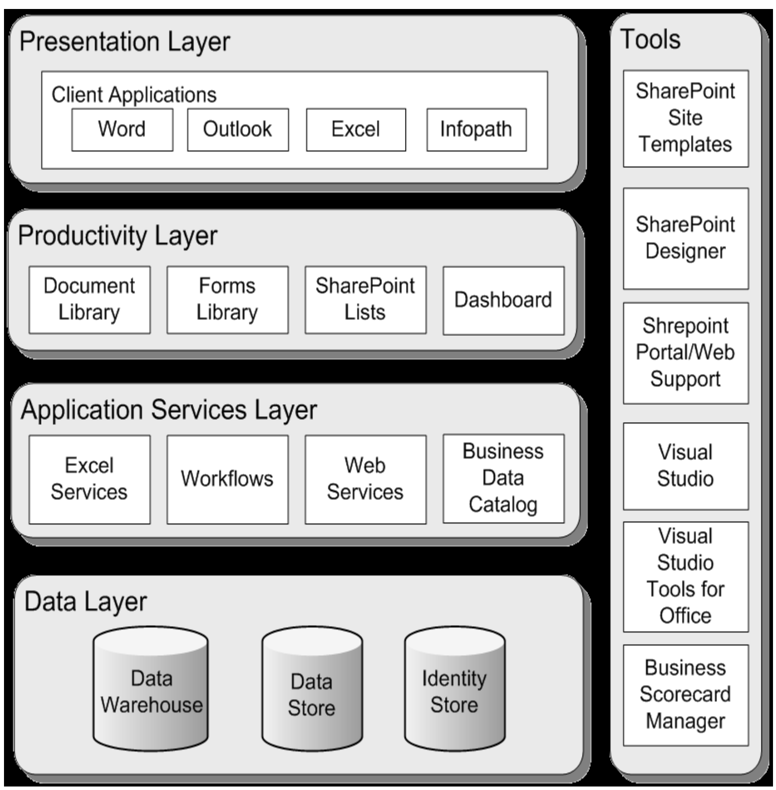


Figura 7.- Arquetipo Aplicaciones OBA

- **Capa de datos:** Esta capa se encarga del almacenamiento de toda la información de la OBA. Tanto de la información utilizada por nuestra aplicación LOB como de la información generada por la OBA.

- **Capa de servicios:** Expone la funcionalidad de nuestra aplicación LOB en forma de servicios para que puedan ser consumidos por Word, Excel, Outlook, etc.

- **Capa de productividad:** Esta capa se añade entre la capa de servicios y la de presentación y se encarga de almacenar y manejar los flujos de trabajo colaborativos en torno a los documentos.

Los componentes que se usan en cada una de estas capas ya existen. Al crear una OBA lo que hacemos es integrarlos y personalizarlos para que resuelvan nuestros problemas. Los principales componentes de una OBA son:

- **Microsoft Office:** Permite crear formularios personalizados, reportes especializados, documentos autogenerados, etc. integrados con la lógica de nuestro negocio. También permite crear plug-ins que integran las acciones y datos importantes de nuestra lógica de negocio dentro de Office.

- **Microsoft SharePoint Services:** Es una plataforma que permite construir aplicaciones web especializadas para un negocio donde se permite la compartición de documentos, información e ideas. WSS contempla escenarios desconectados y soporta la sincronización y el manejo de tareas.

- **Microsoft Office SharePoint Server:** Aumenta las prestaciones de WSS y ofrece herramientas de gestión de contenidos, flujos de trabajo, búsquedas, visualización y edición de ficheros Excel con datos de negocio…

Los problemas que solucionan las OBAs son muy diversos, pero en general se pueden encajar dentro de alguna de estas 3 categorías:

- **Gestión de contenidos de la empresa:** El escenario más común es el uso de MOSS o WSS como una herramienta de gestión de contenidos para documentos de office. Mediante MOSS y WSS se puede gestionar el acceso a la información en función de los roles de los usuarios. MOSS y WSS nos permiten gestionar el flujo de trabajo de un documento y el control de versiones del mismo.

- **Business Intelligence:** Lo más normal es el uso de soluciones basadas en Excel que den solución a problemas específicos de la lógica de negocio de nuestra aplicación LOB.

- **Sistema de mensajería unificado:** En este escenario la OBA se construye para dar soporte a los procesos de comunicación y colaboración proporcionando un sistema de mensajería y notificación de tareas unificadas.

En general, las capacidades o usos que se le pueden dar a las herramientas para construir una OBA son:

- **Llegar a más usuarios**: Utilizar una interfaz conocida, como office, para llegar a más usuarios.

- **Integración de documentos**: Generar documentos de office a partir de aplicaciones LOB.

- **Flujo de documentos:** Gestionar los flujos de control y monitorización en los procesos centrados en los documentos.

- **Interfaz gráfica compuesta**: Gestionan la visualización de varias componentes gráficos dentro de un documento de office o una página de SharePoint.

- **Agrupación de datos:** Permite a los usuarios buscar información originada desde varias aplicaciones LOB distintas. La OBA se encarga de reunir todos estos datos y presentárselos al usuario.

- **Notificaciones y tareas**: Son aplicaciones que utilizan Outlook para gestionar las notificaciones y tareas generadas por una o varias aplicaciones LOB.

En general, aunque las OBAs pueden ser muy distintas unas de otras, existen una serie de consideraciones de diseño que son comunes para todas ellas. Estas consideraciones son:

- Considera usar un intermediario para la integración en lugar de hacerlo directamente: Para llegar a más usuarios es mejor apoyarse en las herramientas como las que proporciona SharePoint para acceder a datos de negocio frente a integrar directamente el acceso a los mismos en un documento office o Excel.

- Usa OpenXML para introducir datos de LOB en documentos: OpenXML es un estándar ECMA por lo que emplearlo en los documentos mejora la interoperabilidad.

- Crea plantillas de documentos LOB para los diseños comunes que vayan a ser reutilizados: Estas plantillas contienen metadatos que permiten enlazar datos de la LOB posteriormente.

- Utiliza MOSS para gestionar la revisión y la aprobación de documentos: MOSS permite realizar esta tarea, por lo que no hay que implementarla. Para procesos más complejos se debe usar Workflow Foundation.

- Implementa el patrón de colaboración para los flujos con interacción de personas: Muchas aplicaciones LOB manejan muy bien los flujos de negocio pero fallan al manejar los flujos de trabajo humanos.

- Considera la sincronización remota de los datos: Los documentos creados o distribuidos deberían estar sincronizados con la aplicación LOB y almacenados para uso futuro.

8.- ARQUETIPO 'APLICACIÓN DE NEGOCIO BASADA EN SHAREPOINT'

SharePoint combina la experiencia de una interfaz de usuario lograda u un potente acceso a datos. Los sitios basados en SharePoint almacenan las definiciones, el tipo de contenidos, los contenidos y la configuración de la aplicación.

La arquitectura normal de una aplicación creada mediante SharePoint se muestra a continuación:

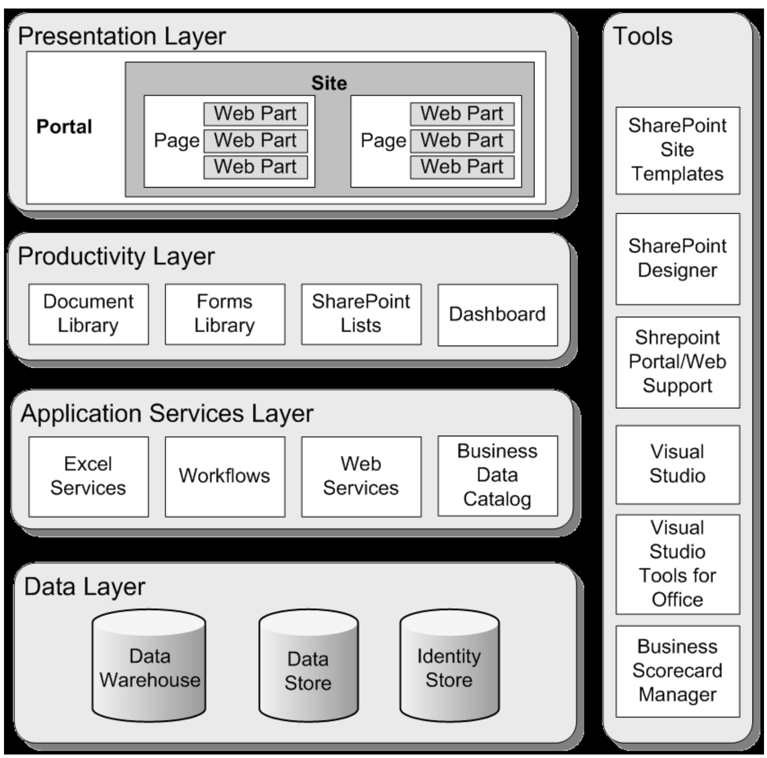


Figura 8.- Arquetipo Aplicación SharePoint

- **Capa de datos:** Esta capa encapsula los mecanismos para almacenar y acceder a los diferentes tipos de datos requeridos por la aplicación.

- **Capa de servicios de aplicación:** Esta capa integra mediante servicios las funcionalidades de otras aplicaciones y los flujos de trabajo de las mismas.

- **Capa de productividad:** Esta capa organiza los documentos subidos por los usuarios, permite crear y publicar reportes y generar un documento con información de múltiples fuentes.

- **Capa de presentación:** Esta capa consiste en una personalización de la interfaz web sobre el esquema básico que proporciona SharePoint mediante Webparts.

MOSS implementa todo lo que necesitamos para crear nuestro sistema de gestión de contenidos. Estas son las características disponibles para explotar SharePoint:

- **Workflow**: MOSS está integrado con Workflow lo que permite a los desarrolladores crear flujos de trabajo sencillos y asociarlos a librerías de documentos en SharePoint.

- **Business Intelligence**: A través de MOSS los usuarios pueden realizar manipulaciones de grandes datos y análisis de los mismos.

- **Gestión de contenidos**: MOSS dispone de herramientas de gestión de contenido web.

- **Búsqueda**: Este servicio permite la recolección de datos, su indexación y su consulta.

- **Catálogo de datos de negocio**: El catálogo de datos de negocio permite exportar datos de la empresa a Web parts, formularios infopath y funciones de búsqueda.

- **OpenXML:** Emplear XML facilita la manipulación de datos desde el servidor.

Cuando se diseña una aplicación SharePoint hay muchas decisiones de diseño que hay que tomar con cuidado. A continuación se presentan las principales recomendaciones:

- **Especializa la interfaz de usuario en función de su Rol:** Proporciona diferentes interfaces gráficas basadas en el rol del usuario. Usa para ello los grupos de seguridad o los públicos objetivos.

- **Integra tu LOB con office:** Integra las herramientas de office que son útiles y específicas para tu solución.

- **Reduce el acoplamiento entre capas:** Usa servicios web para reducir el acoplamiento.

- **Considera la necesidad de sincronización remota:** Todos los documentos creados, actualizados o distribuidos deberían estar sincronizados con el LOB y almacenados para uso futuro.

- **Expón los datos de los LOBs a través de servicios para que sean usados por SharePoint y OBAs:** Exponer los datos de los LOBs a través de servicios permite que office y SharePoint los manipulen y los formateen para el usuario.

CAPÍTULO 11

Arquitectura y Patrones en ‘Cloud-Computing’ PaaS

Existe actualmente mucha literatura sobre qué es ‘Cloud Computing’ y los diferentes tipos de *Cloud*, tanto por modelo de Servicio (*IaaS, PaaS, SaaS*) como por despliegue (Nubes públicas, privadas, híbridas, etc.). Así mismo, disponemos también de gran cantidad de información sobre cada una de las plataformas y tecnologías que ofrecen los diferentes fabricantes/proveedores de nubes.

En nuestro caso (Guía de Arquitectura de Aplicaciones), lógicamente nos interesan especialmente las nubes de tipo PaaS e IaaS, puesto que se trata de definir la arquitectura, diseño e implementación de aplicaciones. No vamos a hacer uso de un producto/servicio ya acabado (SaaS).

Dadas estas premisas, queremos destacar que los objetivos del presente capítulo no se centran, por tanto, en volver a definir qué es ‘Cloud Computing’ ni tampoco en enumerar y explicar los elementos de las tecnologías *Cloud-PaaS* de Microsoft (Plataforma de Servicios de *Windows Azure*), porque aportaríamos poco valor, al estar dicho contenido ya disponible en infinidad de fuentes.

Sin embargo, creemos interesante identificar qué elementos y **patrones de una arquitectura de aplicación** pueden e incluso deben ser diferentes en una **aplicación que vaya a ser desplegada en ‘la nube’**. Así pues, nos centraremos exclusivamente en identificar patrones de Arquitectura y Diseño que encajan bien con la filosofía de ‘Cloud-Computing’ y posteriormente estudiaremos su posible implementación con tecnología Microsoft (.NET y plataforma de servicios de Windows Azure).

1.- ARQUITECTURA DE APLICACIONES EN LA NUBE

Con *Cloud-Computing* siendo una realidad consolidada y con diferentes fabricantes (como *Microsoft*, *Amazon*, *Google*, *IBM*, *Salesforce.com* y un largo etc.) apostando seriamente por ello, las organizaciones están actualmente preguntándose y tratando de evaluar si realmente pueden ayudarles las capacidades y servicios de *cloud-computing*.

Para ello, lógicamente, las empresas y todo tipo de organizaciones quieren analizar el estado actual de sus tecnologías de infraestructura y especialmente de sus aplicaciones, e identificar en qué casos *cloud-computing* puede ser útil o no (evaluando por separado opciones de nubes públicas y de nubes privadas).

En definitiva, se trata de identificar qué aplicaciones pueden beneficiarse de forma sustancial de aspectos como:

- Gran escalabilidad bajo demanda
- Elasticidad en dicha escalabilidad (tanto creciente como decreciente), ideal para aplicaciones con picos de uso, con cualquier periodicidad.
- Despliegues rápidos y sencillos

Una vez identificados estos casos de aplicaciones (nuevas o migraciones de aplicaciones actuales), y seleccionada la tecnología de *cloud-computing* (IaaS o PaaS) a hacer uso (como puede ser la plataforma de Windows Azure) lógicamente, las organizaciones quieren adoptarlo. Para ello hay que realizar un trabajo de:

- Análisis de Migraciones de aplicaciones
- Definir y planificar la estrategia de migraciones

Lo más importante a tener en cuenta con Cloud-Computing es que el cambio hacia ello no se ocasiona normalmente por aspectos técnicos ni de arquitectura. El detonante va a venir normalmente por 'negocio', es decir, si las capacidades de *cloud-computing* (p.e. menor TCO en aplicaciones con picos) se alinean con las necesidades de negocio (necesidades de reducciones de coste, necesidades de mayor agilidad y velocidad en despliegues), entonces es cuando se pasará a la nube.

Una vez situados en este punto (aplicaciones concretas identificadas como parte de la estrategia de Cloud de una organización), entonces es cuando empieza a ser importante pensar en términos de Arquitectura, pues las aplicaciones que encajen con ciertos tipos de estilos arquitecturales serán las que tiendan a encajar con los patrones de migración deseados.

Así pues, identificar aplicaciones que encajan bien con *cloud-computing* es fundamental a la hora de tener éxito en las migraciones.

Desde el punto de vista de ARQUITECTURA, hay varias preguntas interesantes que muchos profesionales se hacen actualmente. Especialmente las siguientes:

- ***'¿Son diferentes las Arquitecturas Lógicas de Aplicaciones en la nube?'***

- ***'¿Debo emplear en la nube la misma arquitectura lógica (mi arquitectura N-Capas, DDD, etc.) que utilizo en mis aplicaciones actuales? '***

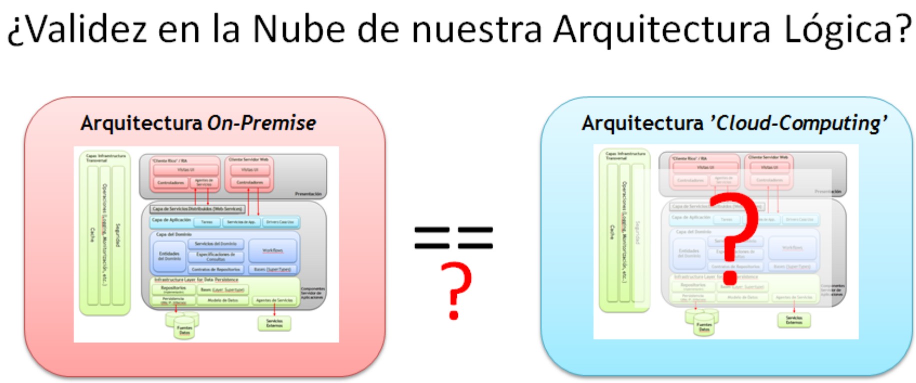


Figura 1.- Validez en la Nube de nuestra Arquitectura Lógica

Bueno, la verdad es que pasa como siempre en nuestro sector, pues la contestación es "depende"…

Ciertamente, podemos migrar una aplicación '*On-Premise*' (en servidores propios) a la nube y mantener el 100% de la misma arquitectura que tenía (por ejemplo nuestra Arquitectura N-Capas DDD) e incluso mantener prácticamente toda la implementación que teníamos en .NET al migrarlo a Windows Azure y SQL Azure. Pero.., **¿debemos migrar directamente o debemos plantearnos arquitecturas y patrones diferentes e incluso diferentes tecnologías de implementación y persistencia de datos?**

Pues la contestación sigue siendo 'depende'. '*Cloud-Computing*' no es otra dimensión o galaxia a la hora de diseñar una aplicación, al final, la arquitectura a diseñar depende también de los objetivos y necesidades de nuestra aplicación.

También, tenemos que pensar en las razones por las que queremos desplegar nuestra aplicación en la nube. 'La nube' (refiriéndonos en este caso a PaaS) es simplemente un nuevo entorno de despliegue que nos ofrece una elasticidad bajo demanda de la escalabilidad de nuestra aplicación y también nos simplifica enormemente las tareas y por lo tanto el coste de despliegue. Sin embargo, por la esencia de '*Cloud-Computing*', en muchos casos vamos a querer 'ir a la nube' porque tenemos nuevos y ambiciosos requerimientos de escalabilidad hasta límites que inicialmente es posible que no conozcamos.

Podemos resumir la contestación a las anteriores preguntas con la siguiente tabla:

Tabla 1.- Diferente Arquitectura y Patrones en Cloud vs. On-Premise

Premisas	Arquitectura y Patrones
A. - **Requerimientos de aplicación similares a una versión de la aplicación *On-Premise*** - **No se requiere una alta escalabilidad** - **Razones para desplegar en la nube son del tipo:** ○ **Facilidad y rapidez en el despliegue** ○ **Pago por uso (OPEX vs. CAPEX)**	La Arquitectura lógica y por lo tanto la mayoría de la implementación (tecnologías) puede ser muy similar a como lo haríamos con una aplicación estándar en nuestros servidores. Siempre pueden surgir pequeñas incompatibilidades a nivel de tecnologías que deberán cambiarse, pero la arquitectura y la mayor parte de la tecnología (.NET) será muy similar a crear una aplicación *On-Premise*.
B. - **Si se requiere una alta escalabilidad**	La Arquitectura lógica debe ser diferente para poder conseguir una dimensión de escalabilidad mucho mayor. Probablemente, en un porcentaje considerable, la implementación (tecnologías necesarias) será diferente y deberemos hacer uso de ciertas tecnologías nativas de la nube que en algunos casos no tenemos disponible en nuestros servidores.

Es ciertamente importante aclarar este punto. La arquitectura física va a ser necesariamente diferente en la nube PaaS que en nuestros servidores, pues existirán elementos físicos diferentes e incluso el despliegue de la aplicación será diferente. Sin embargo, **la arquitectura lógica no debería verse afectada por el entorno de despliegue** (Cloud vs. On-Premise). Si determinamos que requerimos de ciertos patrones de arquitectura y diseño para poder afrontar los objetivos de escalabilidad. Esto debería ser así tanto si el despliegue es *on-premise* (Servidores tradicionales) como si el despliegue es en la nube. Probablemente las tecnologías para conseguirlo no tengan por qué ser iguales, pero los patrones de arquitectura si lo serán.

Así pues, en la mayoría de los casos en los que requerimos cambiar nuestra arquitectura lógica y patrones de diseño para una aplicación 'en la nube', realmente dichos cambios de arquitectura no son ocasionados por 'la nube' en sí, son ocasionados por nuevas necesidades de escalabilidad. Pero ocurre que coincide el hecho de que 'la

nube' (PaaS) nos simplifica la implementación y despliegue de este tipo de aplicaciones escalables.

2.- ESCENARIOS DE ARQUITECTURA EN LA NUBE

Vamos a exponer los anteriores escenarios desde los dos puntos de vista acostumbrados en esta guía, es decir, primero a nivel de arquitectura y patrones y después a nivel de implementación y tecnologías concretas, mostrando su mapeo/relación.

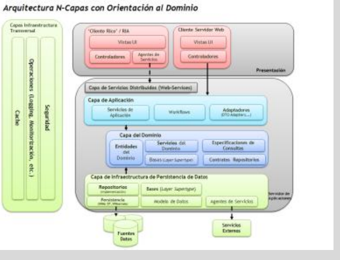
3.- ESCENARIO BÁSICO: MIGRACIÓN DIRECTA DE APLICACIÓN ON-PREMISE A LA NUBE

Este escenario nos permitirá tener un grado muy alto de compatibilidad de nuestra aplicación entre un entorno de despliegue '*on-premise*' (servidores estándar en *datacenters*) y plataformas de '*cloud-computing*'. Sin embargo, este escenario puede no ser válido para ciertos niveles de gran escalabilidad (número muy alto de usuarios concurrentes).

3.1.- Arquitectura Lógica (Escenario Básico)

En el caso de hacer uso de la nube simplemente por razones de facilidad de despliegue, pago por uso (OPEX vs. CAPEX) u otras razones no relacionadas con la escalabilidad, la arquitectura lógica de nuestra aplicación puede ser prácticamente la misma que si desplegamos nuestra aplicación en la nube. Es más, incluso la implementación (tecnologías, .NET y persistencia de datos) sea probablemente casi igual. En nuestro caso (aplicaciones complejas con un volumen importante de lógica de negocio), la arquitectura lógica sería la misma que la analizada en esta guía, es decir, nuestra arquitectura N-Capas DDD:

Puesto que es la misma arquitectura que la utilizada en '*On-Premise*', no vamos a entrar en su análisis. En nuestro caso, se trataría de nuestra misma arquitectura N-Capas DDD.



3.2.- ¿Por qué hacer uso de Windows Azure?

Antes de entrar en la tecnología, queremos resaltar algunas razones de negocio por las que hacer uso de Windows Azure. Para alinear las necesidades de negocio con las capacidades de cloud-computing, las organizaciones deben investigar y evaluar las características de 'cloud' de Windows Azure pero basándose en objetivos de negocio que se quieran conseguir:

Razones de negocio pos las que utilizar Windows Azure

- **Para Reducir Costes**: Las organizaciones deben poder migrar aplicaciones y activos de bases de datos a Windows Azure y tener así un menor TCO (Coste Total de la Propiedad) al pagar solo por los recursos que necesitan y que se están efectivamente utilizando (No pagar en el presente grandes inversiones por lo que en el futuro se pueda necesitar).

- **Para tener un '*Time to Market*' menor**: Las organizaciones pueden hacer uso de las capacidades de despliegues rápidos que ofrece Windows Azure (no solo por el despliegue simplificado sino especialmente por la inmediata disponibilidad de la infraestructura de Windows Azure que se tiene en Internet, no tenemos que comprar, instalar y configurar servidores). Con Windows Azure se ganará tiempo, se incrementará drásticamente la agilidad y en definitiva se disminuyen muchísimo los tiempos de respuesta a los requerimientos de 'negocio'.

- **Para conseguir eficiencia** y mitigar consecuencias de mantenimiento de activos no usados o infrautilizados, las organizaciones pueden hacer uso de Windows Azure para aumentar pero también disminuir los recursos utilizados, mejorando así la eficiencia.

- **Para gestionar correctamente futuras altas demandas de uso**, impredecibles en el presente, y que los usuarios sigan teniendo una experiencia correcta (el sistema sea escalable), las organizaciones pueden hacer uso de la elasticidad de Windows Azure para dinámicamente hacer que sus sistemas escalen según aumenta la demanda.

- **Mover aplicaciones 'clandestinas' a la nube**: En muchas organizaciones existen aplicaciones 'escondidas', es decir, aplicaciones creadas sin la tutela del departamento de IT, en servidores 'clandestinos' y por lo tanto sin ningún nivel de servicio (Bien porque un departamento de negocio lo contrató sin contar con IT o porque son aplicaciones sin la entidad suficiente como para estar bajo el SLA de IT).

El mover esas aplicaciones a Windows Azure hace que pasen a estar en un entorno de Datacenter consistente y seguro que proporciona un SLA mínimo y garantizado de 99.9+. Además hace posible el poder escalar en el futuro dicha aplicación o incluso

disminuir bajo demanda los recursos si tiene muy poco uso o incluso eliminarla de una forma sencilla, todo ello sin prácticamente costes de operaciones.

3.3.- Breve introducción a la plataforma Windows Azure

De forma muy breve y para no obligar a quien no conozca *Windows Azure* a buscar información de otras fuentes para poder entender las siguientes páginas, sintetizamos aquí una breve introducción de *Windows Azure*.

La plataforma *Microsoft® Windows® Azure*™ proporciona procesamiento bajo demanda 'en la nube', donde dicha nube es un conjunto de recursos/servidores interconectados en uno o más *datacenters*. *Windows Azure* es por lo tanto una nube pública e híbrida, de tipo **PaaS** (Plataforma como Servicio) e **IaaS** (Infraestructura como Servicio).

Actualmente, la plataforma de *Windows Azure* está disponible en *datacenters* de Microsoft en Estados Unidos, Europa y Asia. Los Desarrolladores y personal de IT pueden hacer uso de la nube (de una forma muy simplificada) para desplegar y ejecutar sus aplicaciones e incluso para almacenar datos. Así mismo, también las aplicaciones *on-premise* (en *datacenters* propios de la organización) pueden hacer uso remoto de los recursos localizados en la nube de *Windows Azure*, bien porque sea una aplicación de escritorio consumiendo servicios web remotos (*WinForms* ó *WPF*), *RIA* (*Silverlight*), o incluso una aplicación servidor accediendo a recursos en la nube.

La plataforma de *Windows Azure* nos abstrae de los recursos hardware mediante la virtualización. Cada aplicación (Web/Servicios Web) que se despliega a *Windows Azure* se ejecuta en una o más máquinas virtuales (VMs). Esas aplicaciones desplegadas se comportan como si estuvieran en una máquina dedicada, pero pueden compartir con otras VMs recursos físicos como espacio en disco, ancho de banda de red, *cores* de CPU, etc. dentro del mismo host/servidor físico (Como pasa normalmente con la virtualización).

Un beneficio clave de la capa de abstracción sobre el hardware es la portabilidad y escalabilidad. La virtualización de un servicio permite moverlo/clonarlo a cualquier número de servidores físicos del *datacenter*. Al combinar las tecnologías de virtualización con hardware creciente, multi-organización y agregación bajo demanda, Microsoft puede manejar volúmenes de 'economía de escala'. Eso genera una utilización más eficiente de los *datacenters*.

La virtualización en *Windows Azure* nos proporciona también escalabilidad vertical y horizontal.

La **escalabilidad vertical** significa que según aumenta la demanda de usuarios de nuestra aplicación, podemos aumentar el número de recursos de procesamiento de

nuestras máquinas virtuales, como número de *cores* CPU o la memoria asignada a cada VM.

La **escalabilidad horizontal** significa que podemos aumentar/disminuir el número de instancias de VMs (que serán copias/clones de los servicios de nuestra aplicación). Todas las instancias son balanceadas por ***Windows Azure*** (balanceo de carga) a nivel de red, de forma que las peticiones entrantes se distribuyan de forma homogénea entre el número de instancias.

Actualmente, los componentes principales de la ***plataforma de Windows Azure*** son:

- *1. Windows Azure*

- *2. SQL Azure.*

- *3. Windows Azure platform AppFabric*



Figura 2.- Validez en la Nube de nuestra Arquitectura Lógica

1. Windows Azure proporciona las siguientes capacidades:

- Un entorno virtualizado de ejecución de aplicaciones basado en Windows Server (como sistema operativo 'invitado').

- Almacén persistente tanto para datos estructurados (Azure-Storage) como no-estructurado (Blobs) y mensajería asíncrona para arquitecturas escalables.

2.- SQL Azure es básicamente SQL Server proporcionado como servicio en la nube, aunque existe importante valor añadido como alta disponibilidad automáticamente.

3. Windows Azure platform AppFabric proporciona:

- ***Bus de Servicios*** en Internet. Nos ayuda a conectar aplicaciones que estén ejecutándose en nuestros datacenters, en la nube de Windows Azure o en otras nubes, sin importar la topología de red (sin importar si existen firewalls de por medio)

- ***Servicio de Control de Acceso***: Gestiona aspectos de autorización y autenticación para Servicios Web con tokens de seguridad. Sigue la tendencia de 'Orientación a Claims'.

La plataforma también incluye varios servicios de gestión que permiten controlar los anteriores recursos, bien mediante un portal web o por programa mediante un API.

Finalmente se tiene disponible un ***SDK*** y herramientas integradas en Visual Studio de forma que se pueda desarrollar, probar y desplegar todo de forma local en PCs de desarrollo (entorno simulado de Windows Azure llamado '*Local Azure Fabric*') de forma independiente y aislada a la nube. El despliegue a la nube sería el paso final, pero no imprescindible durante la mayor parte del proceso de desarrollo.

Windows Azure está diseñado para abstraer la mayoría de la infraestructura que normalmente es necesaria en servidores tradicionales (servidores, sistema operativo, servicios web, balanceadores de carga, etc.), de forma que los equipos de un proyecto pueden focalizarse solo en construir la aplicación. Aproximadamente, la visión simplificada que se puede tener de Windows Azure (teniendo en cuenta la administración y no solo la aplicación a desplegar) es la siguiente.

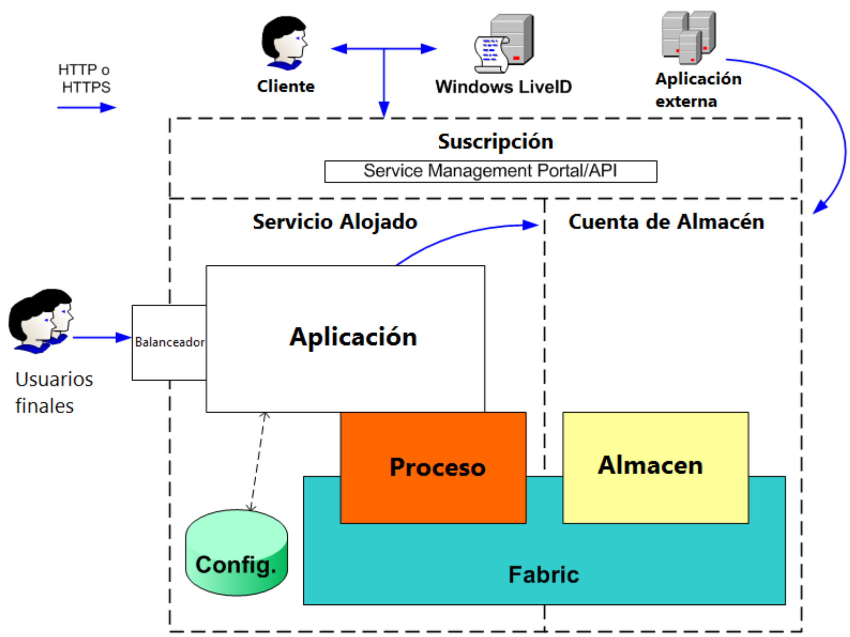


Figura 3.- Validez en la Nube de nuestra Arquitectura Lógica

Los clientes de Windows Azure gestionan sus aplicaciones y almacenes de datos mediante una suscripción que está asociada a una cuenta de Windows Live ID. Esta cuenta es simplemente para la administración de Windows Azure, luego, nuestra aplicación podrá utilizar el sistema de autenticación que consideremos oportuno, para nada tiene por qué ser Windows Live Id, podría ser Membership, orientación a claims basada en Azure Access Control y ADFS 2.0, OpenID, etc.

3.3.1.- Procesamiento en Windows Azure

A una aplicación que se ejecuta en Windows Azure se la denomina 'Servicio Alojado'. Normalmente, un 'Servicio Alojado' contiene diferentes recursos que procesan información e interactúan con el mundo externo (clientes u otras aplicaciones). Los 'Servicios Alojados' están compuestos por roles y actualmente disponemos de dos tipos: Worker role y Web-Role.

Nota:
También disponemos de otro tipo de rol más a un nivel IaaS (Infraestructura como Servicio) denominado VM-Role, donde básicamente es una plantilla/template de máquina virtual que debemos instalar desde cero, incluyendo el sistema operativo (Windows Server 2008 R2) y donde podemos instalar todo el software de base que deseemos, pero también encargarnos de todo el mantenimiento, actualización de partches, nuevas versiones de Sistema Operativo o Service Packs, etc. todo lo cual es hecho automáticamente por Microsoft en los roles Web-Role y Worker-Role..

Sin embargo para conseguir el nivel y productividad de PaaS, es necesario hacer uso de roles de más alto nivel (como el Web rol y Worker rol) donde no es necesario encargarnos de toda la administración e instalación del software de base.

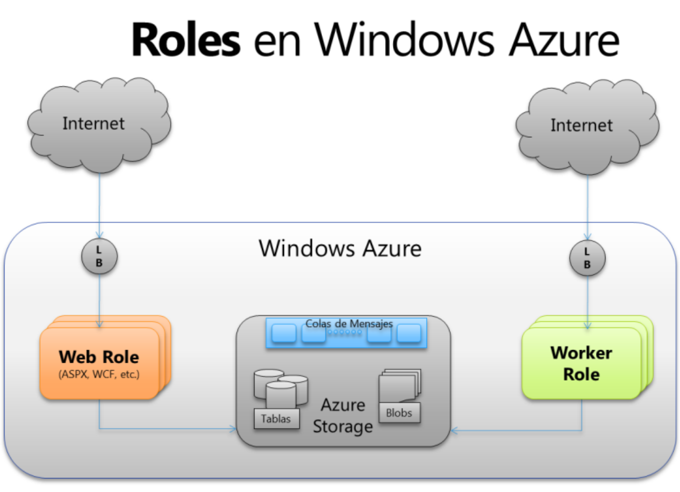


Figura 4.- Roles en Windows Azure

Los 'Worker role' son hosts de propósito general (cada instancia es realmente una VM). Normalmente se utilizan para tareas de ejecución larga que no son interactivas (parecido a cómo funciona un Servicio-Windows). De forma avanzada, se puede incluso alojar en un Worker-role plataformas de aplicaciones completas como la máquina virtual de *Java* y *Apache Tomcat*.

Windows Azure arranca los *Worker-Roles* y de forma parecida a los Servicios-Windows, se están ejecutando de forma continua.

Los *Web-Roles* se pueden ver como un caso concreto de *Worker-Role* pero que ya tiene por defecto IIS instalado.

Normalmente, una instancia web-rol acepta peticiones entrantes HTTP ó HTTPS por los puertos 80 y 443. A estos puertos públicos se les denomina *'endpoints'* públicos. Todos los '*endpoints*' públicos son balanceados a nivel de red, de forma automática.

En los roles *Worker* también se puede hacer uso de puertos TCP como peticiones entrantes, además de HTTP/HTTPS. Las aplicaciones desplegadas en *Web-Roles* pueden implementarse con **ASP.NET**, **WCF** o incluso con cualquier otra tecnología compatible por defecto con **IIS**, por ejemplo *PHP*, puesto que IIS soporta *FastCGI*.

3.4.- Implementación de escenario básico en plataforma Windows Azure

Las aplicaciones de negocio basadas en IIS (bien Web, bien RIA) y SQL Server son muy comunes actualmente y efectúan misiones muy diferentes, desde aplicaciones de misión crítica hasta pequeñas aplicaciones departamentales. En cualquier caso, las siguientes arquitecturas físicas son muy comunes:

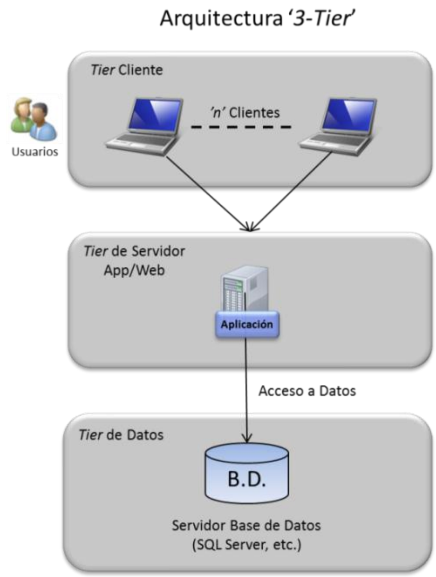


Figura 5.- Aplicación Web/RIA 3-Tier (un único nivel de servidores de aplicación web)

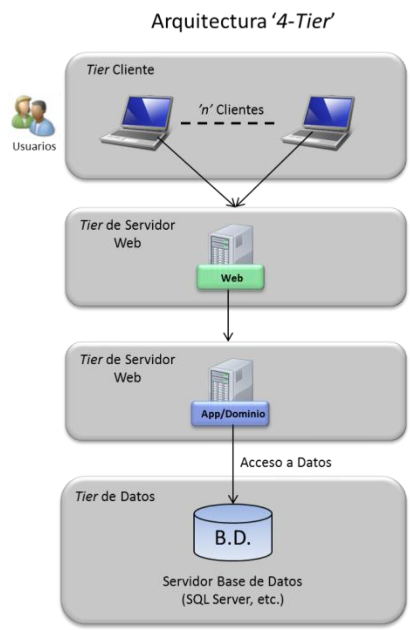


Figura 6.- Aplicación Web/RIA N-Tier (Varios niveles de servidores Web y Negocio)

Las aplicaciones que tienen estos patrones de arquitectura tienden a ser fuertes candidatos para ser migrados a Windows Azure, porque su migración puede ser muy sencilla, obteniendo así los beneficios de Windows Azure, como rápidos despliegues, uso bajo de manda y optimización de recursos, etc.

Debemos resaltar que por el hecho de hacer una 'migración directa' a Windows Azure, no vamos a tener 'mágicamente' una escalabilidad sin límite. La escalabilidad dependerá de cómo esté diseñada la arquitectura de la aplicación y de cómo esté desarrollada e incluso es posible que si simplemente migramos de una forma directa, tengamos techos de escalabilidad inamovibles a no ser que se cambie la arquitectura de la aplicación e incluso las tecnologías (p.e. cambio de SQL Azure a Azure Storage).

En cualquier caso, en este escenario estamos hablando de migraciones sencillas/directas de arquitecturas actuales en servidores pasándolo directamente a Windows Azure. A este caso es a lo que denominamos en esta guía como "*Escenario básico de aplicación en plataforma Windows Azure*".

Normalmente, el objetivo en este escenario es migrar aplicaciones actuales a Windows Azure con el mínimo número posible de cambios en nuestra aplicación.

Así pues, en una migración casi directa de una aplicación '*on-premise*' a Windows Azure, los cambios tecnológicos no serán muy grandes, y los que sean necesarios, serán pequeños cambios muy transparentes.

Este escenario nos permite tener un grado muy alto de compatibilidad entre la versión de nuestra aplicación '*On-Premise*' (a desplegar en un entorno de servidores de aplicaciones ***Windows Server***) y la versión de nuestra aplicación para ***Windows Azure***.

El mapeo entre las diferentes o mismas tecnologías es el siguiente:

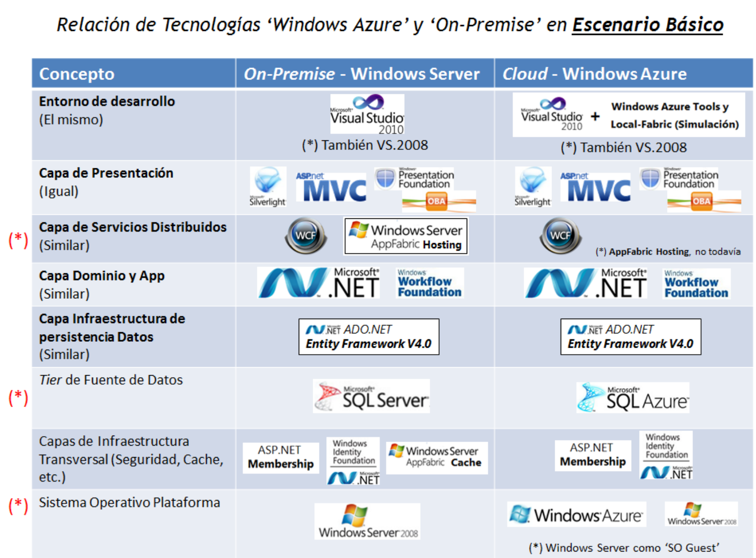


Figura 7.- Relación de Tecnologías 'Windows Azure' y 'On-Premise' en Escenario Básico

Resaltamos con un "(*)" las tecnologías o áreas que requerirán ciertos cambios, aunque sean bastante leves y transparentes.

3.5.- Pasos para migrar Aplicación ejemplo NLayerApp a Windows Azure (Escenario Básico en la nube)

En este apartado mostramos los pasos necesarios para migrar nuestra aplicación ejemplo *NLayerApp* y pueda ser ejecutada en la plataforma de *Windows Azure*.

El caso de esta migración coincide con el escenario básico anteriormente comentado. Es decir, hacemos uso de la misma arquitectura N-Capas DDD y prácticamente de las mismas tecnologías, pues los únicos cambios necesarios son los siguientes.

Cambios requeridos para migración básica a Windows Azure

- **1.- Migración de base de datos de *SQL Server* a *SQL Azure***
    - Es transparente de cada al desarrollo y componentes *.NET* de acceso a datos.
    - En nuestro código simplemente cambiaremos el string de conexión apuntando al nombre DNS de la nueva B.D. de SQL Azure.
- **2.- Adición de proyecto de configuración de *Windows Azure* al '*Solution*' de *Visual Studio*.**
    - Este proyecto solo contiene dos ficheros XML de configuración
- **3.- Adición de código de gestión e instrumentación de Rol de *Windows Azure* en proyectos de *Servicios WCF* y proyectos Web.**
    - Este es el único cambio un poco más intrusivo con respecto a nuestro código anterior. Sin embargo es muy poco código a añadir.
    - Es necesario para poder instrumentar nuestro proyecto como rol de *Windows Azure* (*Rol Web* o *Rol Worker*).

Los pasos detallados de migración básica de nuestra aplicación ejemplo NLayerApp a Windows Azure se explican a continuación.

3.5.1.- Migración de Base de Datos SQL Server

Este paso es bastante transparente. Podemos sustituir nuestra base de datos en SQL Server por una base de datos similar en SQL Azure. El proceso de migración de esquema y datos de una base de datos desde SQL Server a SQL Azure es bastante sencillo (Aunque tenemos varias posibilidades para hacerlo).

NOTA: Existen algunas pequeñas incompatibilidades entre bases de datos de *SQL Server* y *SQL Azure*. En la mayoría de los casos no nos veremos afectados, pero hay que asegurarse y analizar que no existen problemas. Para más información sobre la compatibilidad entre bases de datos y *Transact-SQL* de *SQL Server* y *SQL Azure*, ver:
http://msdn.microsoft.com/en-us/library/ff394115(v=MSDN.10).aspx

MIGRACION DE ESQUEMA DE B.D.

Las opciones para migrar el esquema de la B.D. desde SQL Server a SQL Azure son:

- **1.- Generación de SQL Scripts con el '*SQL Server Management Studio*' de *SQL Server 2008 R2*** el cual nos genera el código Transact-SQL compatible con SQL Azure. En la tarea de 'Exportar-Scripts' hay una nueva opción que especifica que lo queremos generar de forma compatible con SQL-Azure:

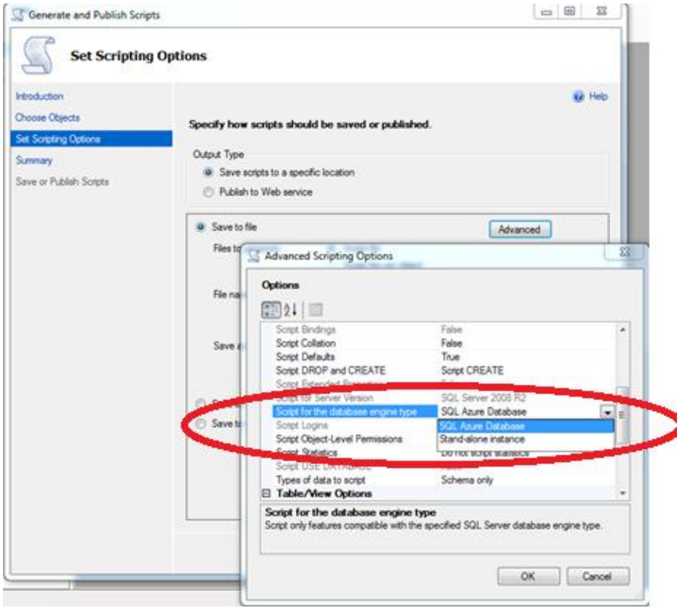


Figura 8.- Podemos especificar que lo queremos generar compatible con SQL-Azure

- **2.- Exportación de paquete DACPAC (Data Tier Packages)**, con *'SQL Server Management Studio'* de *SQL Server 2008 R2* o *Visual Studio 2010* e importación de paquete con '*SQL Server Management Studio*' de SQL Server R2 conectado a servidor lógico de SQL Azure:

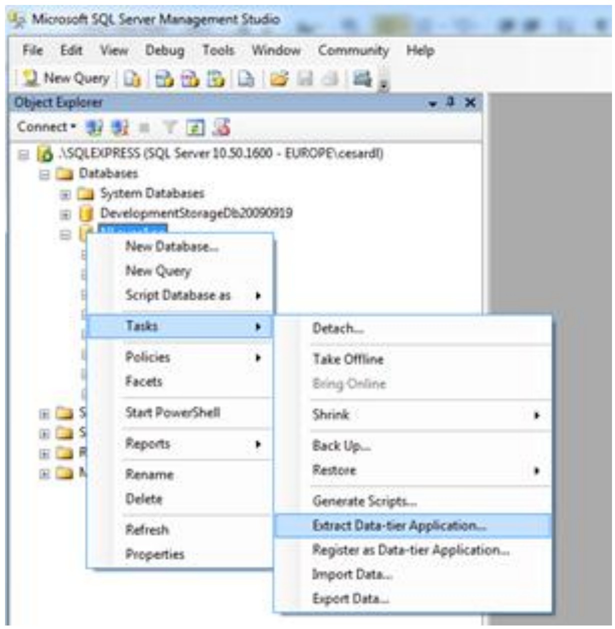


Figura 9.- Exportación de paquete DACPAC desde B.D. SQL Server

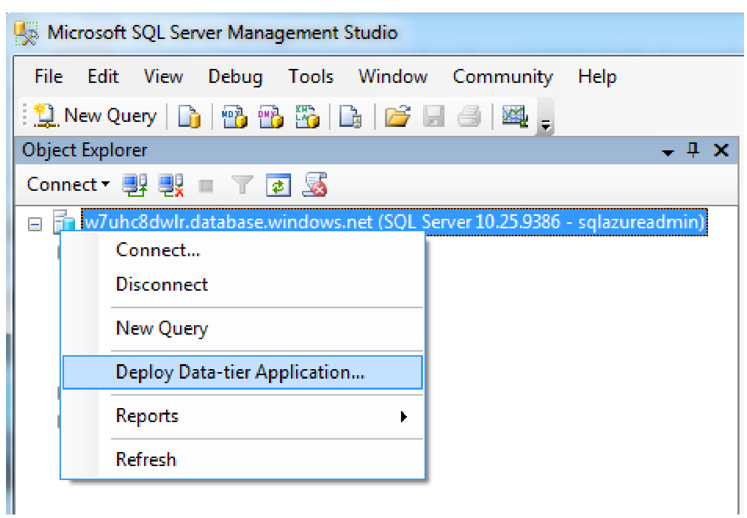


Figura 10.- Importación de paquete DACPAC a SQL Azure

MIGRACION DE DATOS Y ESQUEMA DE B.D.

Creación de B.D. y configuración inicial de SQL Azure

- Lo primero que debemos hacer es crear una base de datos vacía en SQL Azure. Para ello, lo más fácil es crearla desde el portal de SQL Azure:

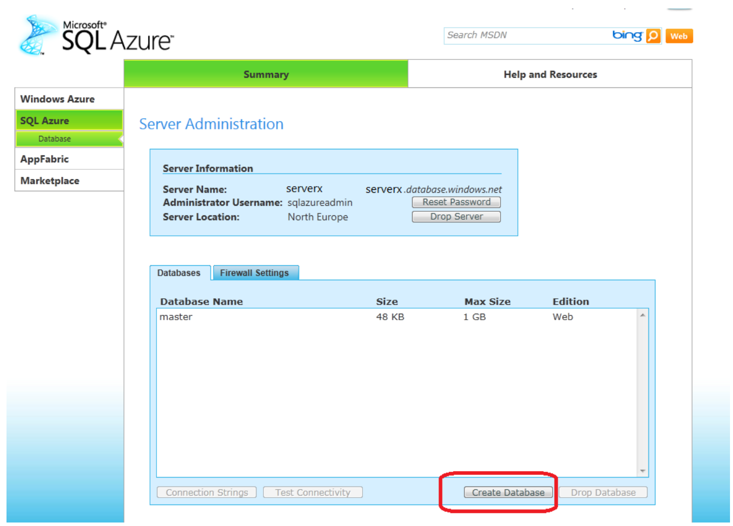


Figura 11.- Lo más fáciles crear la BBDD desde el portal de SQL Azure

En nuestro caso, nuestra base de datos 'NLayerApp', y con tamaño de 1GB nos es suficiente:

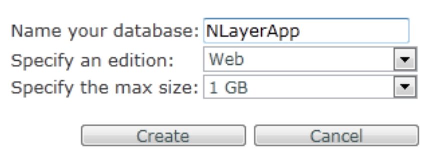


Figura 12.- Definimos nobre y tamaño de la BBDD

Con lo que aparecerá creada de forma similar a esta:

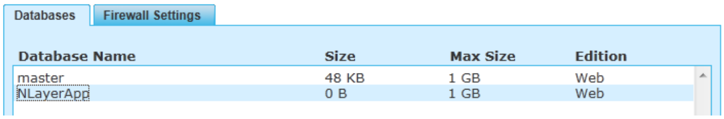


Figura 13.- Por fin está creada la BBDD

- Después, si queremos poder acceder a la base de datos desde aplicaciones cliente remotas (p.e. nuestro PC de desarrollo), debemos abrir el firewall de SQL Azure para que permita accesos externos a cualquier IP, añadiendo una regla de firewall en SQL Azure, de la forma siguiente:

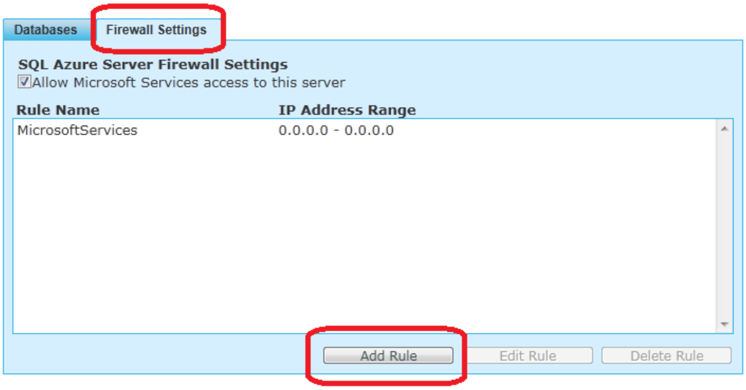


Figura 14.- Debemos añadir una regla que abra el firewall para poder acceder en remoto

Y crear una regla (sólo mientras se desarrolla y prueba, normalmente en producción eliminar esta apertura del firewall) que permita a cualquier IP acceder de forma remota a nuestras bases de datos de nuestro servidor lógico de SQL Azure:



Figura 15.- Configuración que permite el acceso remoto a cualquier IP

De lo contrario, si por ejemplo queremos conectarnos a nuestra base de datos desde *SQL Server Management Studio* o desde el *debugging* de nuestra aplicación en local, obtendremos el siguiente error/excepción:

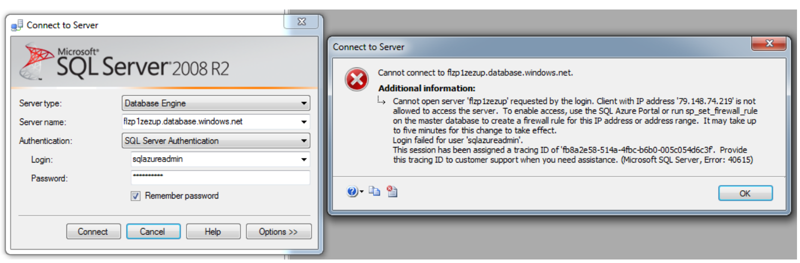


Figura 16.- Error que obtendríamos si no abrimos el Firewall

Migración de B.D. a SQL Azure

Para migrar los datos de una base de datos en SQL Server 'on-premise' a SQL Azure, tenemos también varias opciones:

- **1.- BCP** (Utilidad de línea de comandos), donde de una forma manual podemos migrar los datos. Es útil para sistemas en los que queremos lanzar un Script de migración de datos de una forma periódica. Para conocer cómo realizar esta migración de datos con BCP, ver este *post*:

  http://blogs.msdn.com/b/cesardelatorre/archive/2010/06/04/importing-exporting-data-to-sql-azure-databases-using-bcp-and-sql-scripts.aspx

- **2.- SSIS** (*SQL Server Integration Services*): SSIS tiene capacidades mucho más avanzadas de tipo ETL (*Extract, Transform, Load*). Es muy sencillo realizar un paquete de exportación/importación contra *SQL Azure* similar al siguiente:

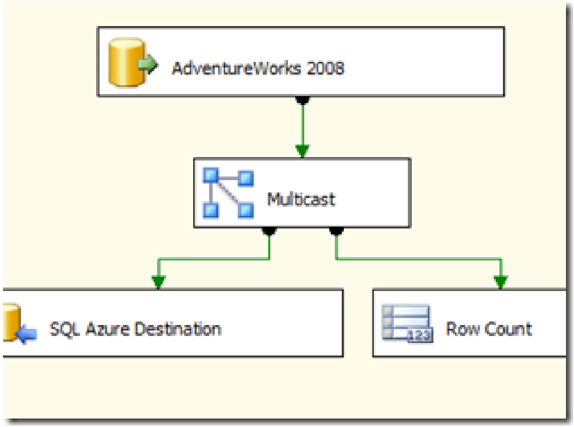


Figura 17.-

- **3.- SQL Azure Data Sync:** Actualmente (en beta) Está basado en *Microsoft Sync Framework 2.0 SDK* y *Microsoft Sync Framework Power Pack for SQL Azure*. Sin embargo, esta tecnología va mucho más allá, ofrece capacidades de sincronización de datos, no solo de migración:

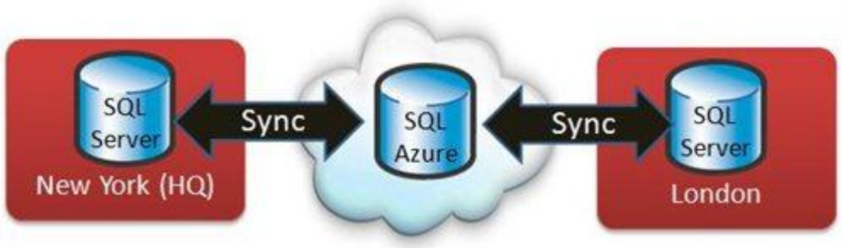


Figura 18.- SQL Azure Data Sync también ofrece capacidades de sincronización de Datos

- **4.- SQL Azure Migration Wizard**: Es probable, actualmente, la opción más cómoda y utilizada para realizar migraciones sencillas de bases de datos, porque es capaz de realizar la migración tanto del esquema de la base de datos como también de los datos:

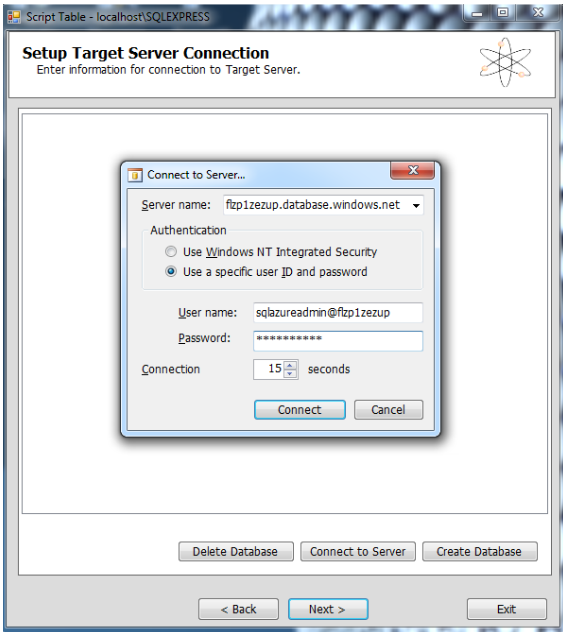


Figura 19.- Para migraciones sencillas es más cómodo usar SQL Azure Migration Wizard

SQL Azure Migration Wizard está disponible en: **http://sqlazuremw.codeplex.com/**
La migración de nuestra base de datos de *SQL Server* de la aplicación ejemplo, llamada '**NLayerApp**', se puede migrar a *Windows Azure* por cualquiera de los métodos anteriores. Probablemente el más sencillo y directo es precisamente mediante ***SQL Azure Migration Wizard***.

Administración de B.D. a SQL Azure

Actualmente disponemos de dos caminos para poder administrar nuestra base de datos en la nube de SQL Azure:

- ***1.- SQL Server Management Studio de SQL Server 2008 R2***:

  Esta opción es la más potente y simplemente necesitamos instalar en nuestra máquina de desarrollo/administración el '***SQL Server Management Studio de SQL Server 2008 R2***'. Una vez instalado, podemos conectarnos remotamente a nuestra base de datos de SQL Azure, de forma similar a la siguiente:

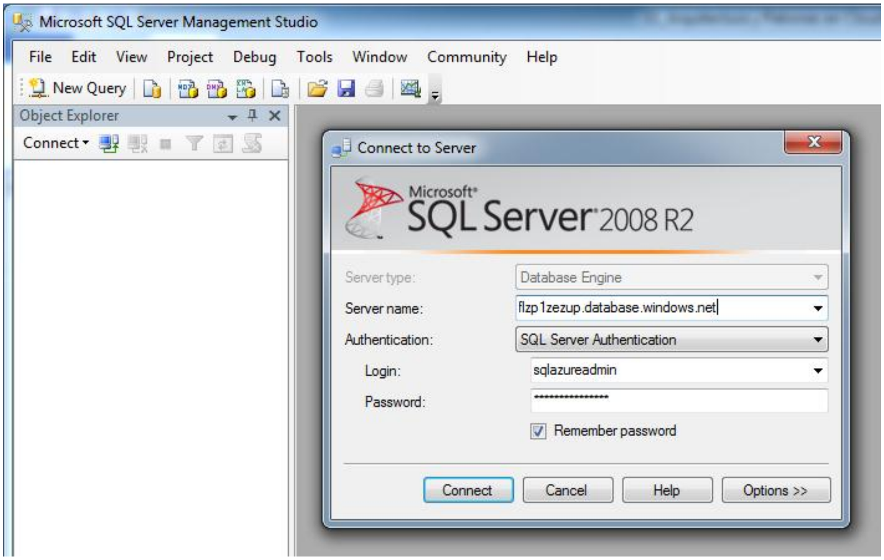


Figura 20.-

Y posteriormente, administrar nuestra base de datos:

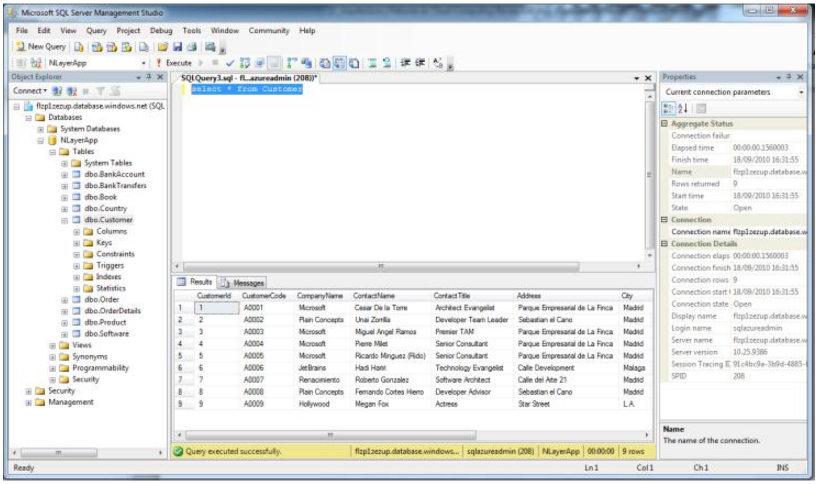


Figura 21.-

- *2.- Microsoft Project Code-Named "Houston" (Web/RIA)*:

La segunda opción es muy innovadora y consiste en utilizar una nueva aplicación de administración de SQL Azure (actualmente todavía en BETA).

Lo novedoso de esta herramienta es que es Web/RIA (cliente Silverlight), por lo que al estar nuestra base de datos en Internet, desde esta herramienta podemos administrar nuestra base de datos con cualquier PC y navegador, sin necesidad de tener instalado ningún software de administración de SQL Server. Cualquier máquina, con un navegador y el *plugin* de ***Silverlight*** instalado es suficiente para hacer uso de ello. Algunas imágenes de 'Houston':



Figura 22.-

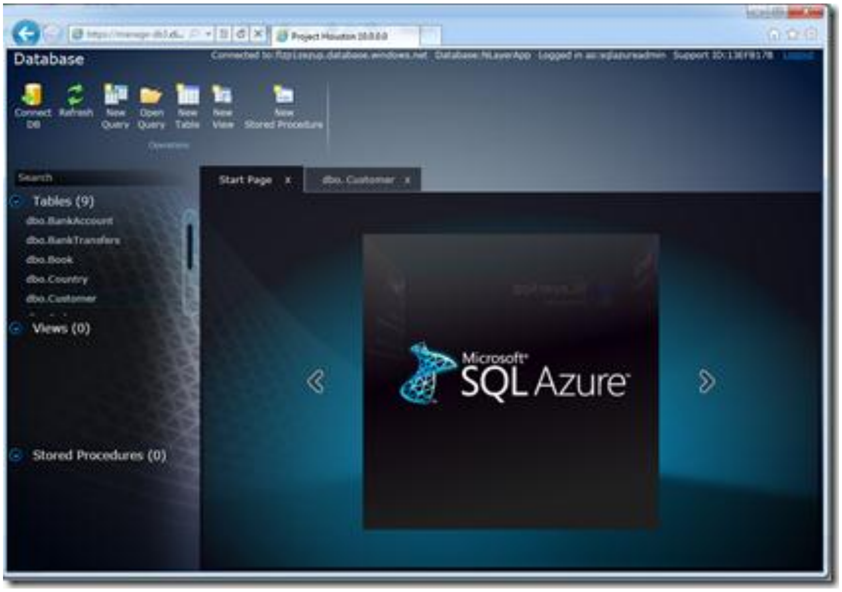


Figura 23.-

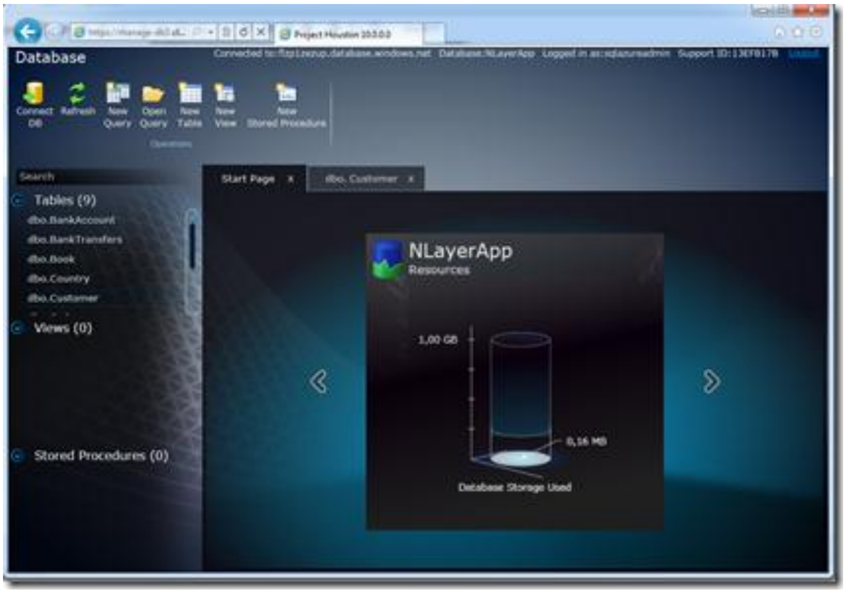


Figura 24.-

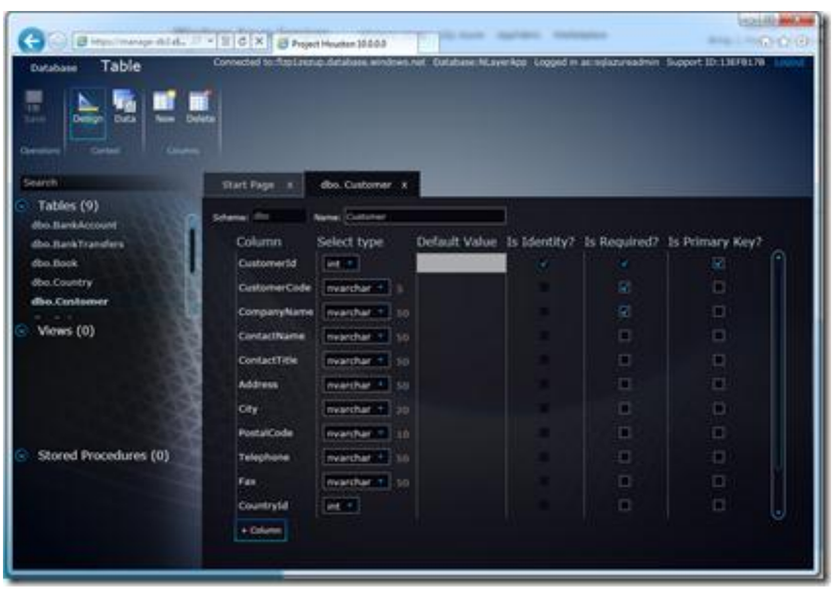


Figura 25.- Administración de Tablas

3.5.2.- Cambio de cadena de conexión de ADO.NET / EF

Este paso es extremadamente sencillo. Simplemente, puesto que ahora nuestra B.D. está en SQL Azure, debemos cambiar la cadena de conexión que utilizamos en ADO.NET / EF.

Por ejemplo, en nuestra aplicación *NLayerApp*, en el fichero de configuración .config de los servicios WCF, deberíamos cambiar lo siguiente:

```
STRING DE CONEXIÓN DE EF PARA SQL SERVER

   <add name="MainModuleContext"
connectionString="metadata=res://*/Model.MainModuleDataModel.csdl|res://
*/Model.MainModuleDataModel.ssdl|res://*/Model.MainModuleDataModel.msl;p
rovider=System.Data.SqlClient;provider connection string=&quot;Data
Source=.\SQLEXPRESS;Initial Catalog=NLayerApp;Integrated
Security=True;MultipleActiveResultSets=True&quot;"
providerName="System.Data.EntityClient" />
```

```
STRING DE CONEXIÓN DE EF PARA SQL AZURE

   <add name="MainModuleContext"
connectionString="metadata=res://*/Model.MainModuleDataModel.csdl|res://
*/Model.MainModuleDataModel.ssdl|res://*/Model.MainModuleDataModel.msl;p
rovider=System.Data.SqlClient;provider connection
string=&quot;Server=tcp:KKuhc8dwlr.database.windows.net;Database=NLayerA
pp;User
ID=sqlazureadmin@KKuhc8dwlr;Password=mipass@word1;Trusted_Connection=Fal
se;Encrypt=True;MultipleActiveResultSets=True&quot;"
providerName="System.Data.EntityClient" />
```

Como se puede apreciar, **básicamente el único cambio a realizar es poner el nombre DNS del servidor SQL-AZURE** así como las credenciales estándar de SQL Server.

Si no estuviéramos utilizando EF, sino simplemente ADO.NET, el cambio sería similar, solamente nombre de servidor y credenciales de seguridad.

> **Nota:** El usuario y *password* a utilizar (credenciales SQL Server estándar) son necesarios, pues actualmente *SQL Azure* soporta solamente seguridad estándar de *SQL Server* y no seguridad integrada con AD. Algo por otra parte lógico puesto que no tenemos soporte directo de AD en *Windows Azure*..

3.5.3.- Migración de proyectos en *hosting* de IIS a Azure

Migración de *Hosting* de Servicio WCF en IIS/Cassini a Azure WCF WebRole

El migrar el código del Servicio WCF de hosting de la aplicación ejemplo a un nuevo proyecto creado de tipo WCF WebRole de Windows Azure, es muy sencillo.

Los pasos son los siguientes:

- Crear un proyecto de Windows Azure (p.e. llamado '**AzureServices'**) con un role de tipo **WCF Service WebRol** (p.e. llamado '**WCFWebRole'**). Deberemos tener algo similar a lo siguiente con la definición de los roles de Azure:

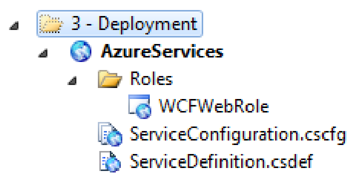


Figura 26.-

- Y con respecto al proyecto WCF, un proyecto vacío/estándar de *WCFWebRole* similar al siguiente:

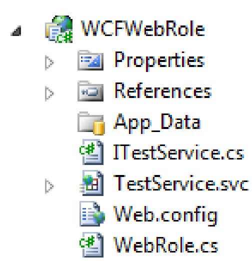


Figura 27.-

- Este proyecto WCF, por ahora, puede estar situado en cualquier parte de nuestro *Solution* de la aplicación *NLayerApp*. Finalmente, lo moveríamos dentro de la carpeta '1.2 Distributed Services'.

- Copiar los ficheros de definición de nuestro proyecto de *hosting* del servicio WCF original de NLayerApp, es decir, los ficheros:

- **crossdomain.xml** (Para permitir accesos de Silverlight desde otros dominios DNS)

- **MainModule.svc** (Fichero de acceso al servicio WCF de nuestro módulo)

- **web.config** original del servicio WCF de NLayerApp, cambiando el código XML de '<system.diagnostics>' por el siguiente que hace uso de Azure.Diagnostics en lugar de *listeners* estándares de .NET-Windows:

```
<system.diagnostics>

      <sharedListeners>
        <add type="Microsoft.WindowsAzure.Diagnostics.Diagno
  sticMonitorTraceListener, Microsoft.WindowsAzure.Diagnosti
  cs, Version=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3
  856ad364e35"
              name="AzureDiagnostics">
                  <filter type="" />
        </add>
      </sharedListeners>

      <trace>
        <listeners>
          <add name="AzureDiagnostics" />
        </listeners>
      </trace>

      <sources>
        <source name="NLayerApp" switchValue="Error">
          <listeners>
            <add name="AzureDiagnostics" />
          </listeners>
        </source>
      </sources>

  </system.diagnostics>
```

- Añadir las referencias a los *assemblies* que utiliza el servicio WCF de nuestro módulo:

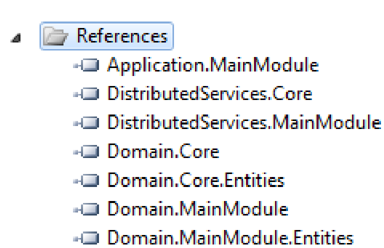


Figura 28.-

- Eliminar el “<serviceDiscovery/>” del web.config, pues en Azure no utilizaremos WS-Discovery al no ser una LAN con *broadcast* local.

- Añadir la línea siguiente dentro de la sección <system.serviceModel>

    - <serviceHostingEnvironment multipleSiteBindingsEnabled="true" />

- Eliminar la línea siguiente relativa también a WS-Discovery:

    - <endpoint name="DiscoveryEndpoint" listenUriMode="Explicit" kind="udpDiscoveryEndpoint" />

- **Comprobar el cambio de String de Conexión EF de base de datos**: Si no lo hemos hecho anteriormente, debemos hacerlo ahora. En el string de conexión de Entity Framework (Web.config de nuestro proyecto *hosting* de WCF en Azure) simplemente debemos cambiar el nombre de servidor de SQL Server por el nombre del servidor lógico de SQL azure, así como especificar las credenciales de acceso a SQL Azure (Usuario y Password).

```
<?xml version="1.0"?>
<configuration>
        <connectionStrings>
    <!-- (SQL AZURE) -->
    <add name="MainModuleContext"
connectionString="metadata=res://NLayerApp.Infrastructure.Data.MainModule/Mo
del.MainModuleDataModel.csdl|res://NLayerApp.Infrastructure.Data.MainModule/
Model.MainModuleDataModel.ssdl|res://NLayerApp.Infrastructure.Data.MainModul
e/Model.MainModuleDataModel.msl;provider=System.Data.SqlClient;provider
connection
string=&quot;Server=tcp:w7xxxxxxxx.database.windows.net;Database=NLayerApp;U
ser
ID=sqlazureadmin@w7xxxxxxxx;Password=mipassword;Trusted_Connection=False;Enc
rypt=True;MultipleActiveResultSets=True&quot;"
providerName="System.Data.EntityClient" />
    <!-- (SQL Server Express) -->
    <!-- <add name="MainModuleContext"
connectionString="metadata=res://NLayerApp.Infrastructure.Data.MainModule/Mo
del.MainModuleDataModel.csdl|res://NLayerApp.Infrastructure.Data.MainModule/
Model.MainModuleDataModel.ssdl|res://NLayerApp.Infrastructure.Data.MainModul
e/Model.MainModuleDataModel.msl;provider=System.Data.SqlClient;provider
connection string=&quot;Data Source=.\SQLEXPRESS;Initial
Catalog=NLayerApp;Integrated
Security=True;MultipleActiveResultSets=True&quot;"
providerName="System.Data.EntityClient" /> -->
        </connectionStrings>
```

Migración de *Hosting* de Web-Silverlight en IIS/Cassini a Azure WebRole

Para hacer esta migración podemos bien modificar nuestro proyecto original de Silverlight o bien crear un nuevo proyecto WebRole de Azure y mover el código de Silverlight a este nuevo proyecto. Elegimos la segunda opción (aunque la otra también es factible). Para ello, creamos un proyecto nuevo WebRole similar al siguiente:

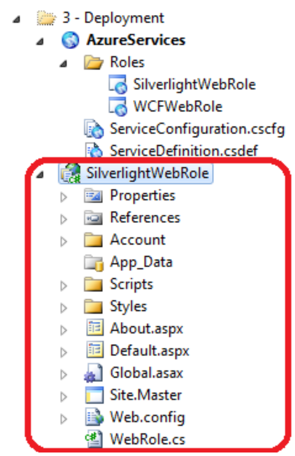


Figura 29.-

- **Adición y cambios de ficheros de Silverlight:** Borramos todas las páginas ejemplo del proyecto, Styles, Scripts, Account, Global.asax, etc. y añadimos los ficheros del proyecto hosting/IIS original de Silverlight, llamado '*Silverlight.Client.Web*':

    - **Web.config**: Añadimos el Web.config de nuestro proyecto original de Silverlight de NLayerApp (realmente es un web.config prácticamente vacío) y lo único que debemos añadirle es el XML relacionado a los Diagnosticos de Windows Azure:

```
    <system.diagnostics>
     <trace>
       <listeners>
         <add type="Microsoft.WindowsAzure.Diagnostics.Diagnost
icMonitorTraceListener, Microsoft.WindowsAzure.Diagnostics, Ve
rsion=1.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e3
5"
           name="AzureDiagnostics">
           <filter type="" />
         </add>
       </listeners>
     </trace>
   </system.diagnostics>
```

Nos quedará un Web.config similar al siguiente:

```
<?xml version="1.0"?>

<configuration>
    <system.web>
        <compilation debug="true" targetFramework="4.0" /
>
    </system.web>

  <system.diagnostics>
    <trace>
```

```
            <listeners>
              <add type="Microsoft.WindowsAzure.Diagnostics.Dia
gnosticMonitorTraceListener, Microsoft.WindowsAzure.Diagn
ostics, Version=1.0.0.0, Culture=neutral, PublicKeyToken=
31bf3856ad364e35"
                name="AzureDiagnostics">
                <filter type="" />
              </add>
            </listeners>
          </trace>
        </system.diagnostics>

      </configuration>
```

- Copiar los ficheros **Silverlight.Client.Web.html** y **Silverlight.js** y establecer a la página **Silverlight.Client.Web.html** como página por defecto en el arranque (Para probar en *debugging*, etc.).

- Añadir la relación entre nuestro proyecto de hosting WebRole de Azure y nuestro proyecto de aplicación Silverlight. Para ello, abrir las propiedades de nuestro proyecto **SilverlightWebRole** y en el 'Tab' 'Silverlight Applications' añadir nuestro proyecto existente '***Silverlight.Client***':

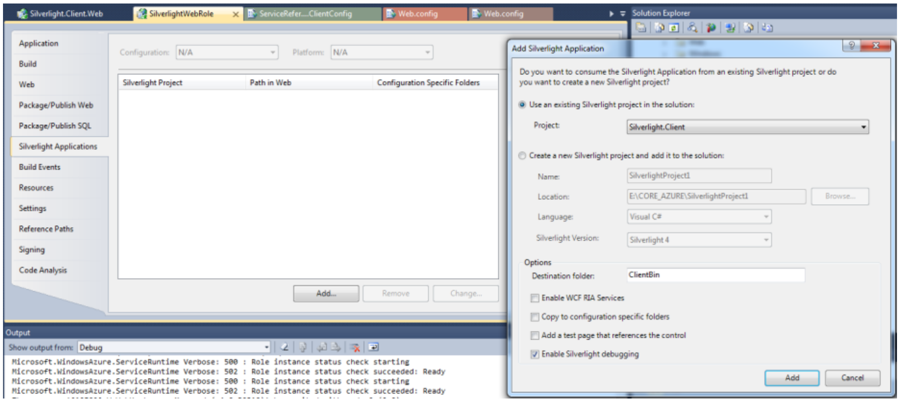


Figura 30.-

Quedando así:

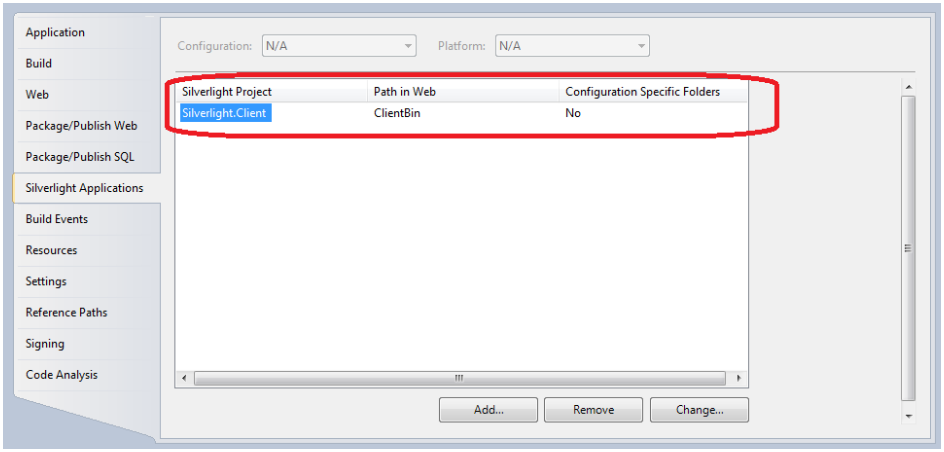


Figura 31.-

Provocando esta acción que se incluya el .XAP de Silverlight en nuestro proyecto *hosting* WebRole:

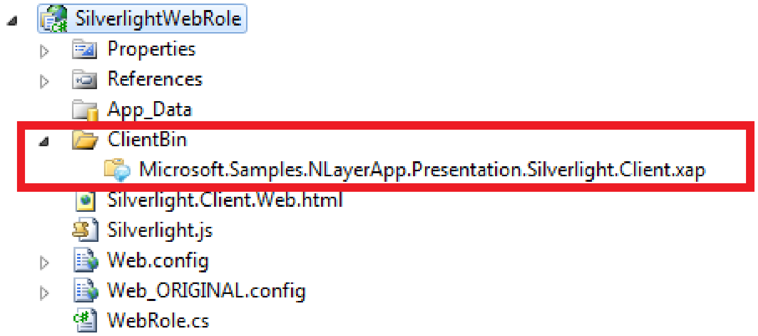


Figura 32.-

- **Cambio de URLs de Servicios WCF a consumir:** En el proyecto de Silverlight llamado **Silverlight.Client** (no en el de *hosting* WebRole de Azure con el que estábamos antes), modificar la configuración de los *bindings* clientes de WCF (en el fichero **ServiceReferences.ClientConfig**) para que consuma ahora los servicios WCF alojados en Windows Azure.

Puesto que mi servicio WCF cuando lo ejecuto en modo prueba con Windows Azure (Azure Development Fabric), es este:

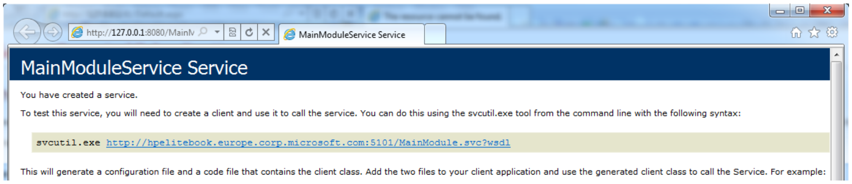


Figura 33.-

Vemos que está en la dirección **http://127.0.0.1:8080** (o el puerto TCP elegido en las propiedades 'Web' del proyecto WcfWebRol, lo cual cambia la configuración de los proyectos/ficheros de Windows Azure), así pues deberemos cambiarlo en el fichero **ServiceReferences.ClientConfig** para que quede algo así:

```
<configuration>
  <system.serviceModel>
    <bindings>
      <basicHttpBinding>
        <binding name="BasicBindingForSilverlightClients" maxBu
fferSize="2147483647"
          maxReceivedMessageSize="2147483647">
          <security mode="None" />
        </binding>
      </basicHttpBinding>
    </bindings>
    <client>
      <!-- ADDRESS FOR LOCAL AZURE DEVELOPMENT FABRIC-->
      <endpoint address="http://127.0.0.1:8080/MainModule.svc/b
asic"
        binding="basicHttpBinding" bindingConfiguration="BasicB
indingForSilverlightClients"
        contract="ServiceAgent.IMainModuleService" name="BasicB
indingForSilverlightClients" />
    </client>
    <extensions />
  </system.serviceModel>
</configuration>
```

Teniendo en cuenta que cuando queramos subirlo a la nube real de Windows Azure en Internet, deberemos cambiar la IP y puerto a las realmente utilizadas en Windows Azure en Internet.

- Así pues, en este punto, si ejecutamos nuestra aplicación en modo debugging (y teniendo como arranque por defecto el proyecto de configuración de Windows Azure), nuestra aplicación se desplegará y ejecutará en el entorno local '**Windows Azure Development Fabric**' (aunque si estaremos accediendo a la B.D. real de SQL Azure), como podemos ver a continuación:

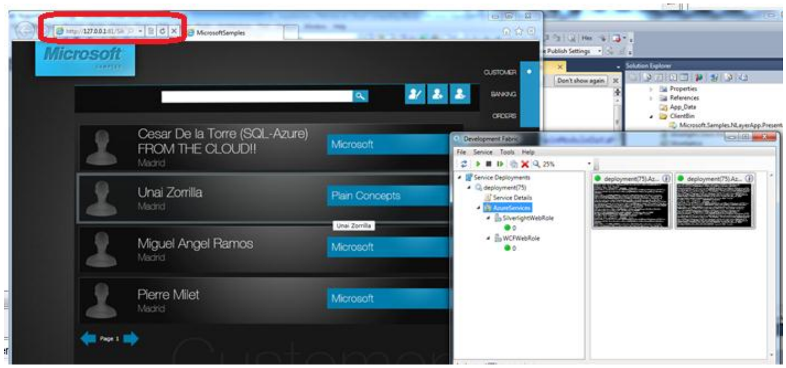


Figura 34.-

Se puede observar que esta ejecución de nuestra aplicación la estamos haciendo en local, basada en la simulación de '*Windows Azure Dev-Fabric*'

3.5.4.- Despliegue en la nube de Windows Azure en Internet

Este cambio de ejecución desde el entorno simulado de Windows Azure (*Windows Azure Development Fabric*) a la nube real de Windows Azure, es muy sencillo.

Configuración de puertos TCP a utilizar en la nube

- Antes de desplegarlo, debemos confirmar/configurar los puertos TCP que se utilizarán en la nube:

    - Confirmar/cambiar que el puerto de ejecución de nuestro WebRole de Silverlight para que se ejecute en el puerto 80. Comprobarlo en las propiedades del *WebRole* en el tab *Endpoints*:

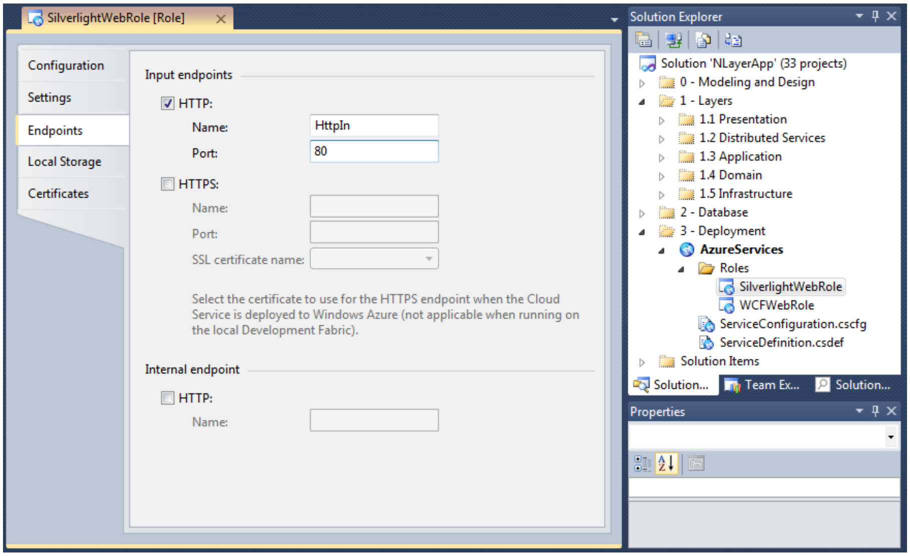


Figura 35.-

- Confirmar también cual es el puerto TCP que está utilizando nuestro servicio WCF en el WCFWebRole. En nuestro caso estamos utilizando el puerto 8080.

Cambiar la URL de consumo de servicio WCF

- Una vez despleguemos nuestro servicio WCF a la nube real de Windows Azure, la URL de nuestro servicio WCF estará basada en la URL de nuestro servicio de Windows Azure, en mi caso completo lo creé como '**http://nlayerazure.cloudapp.net/**', por lo que la URL del *binding* para consumir el servicio WCF debería ser **http://nlayerazure.cloudapp.net/:8080** en lugar de **http://127.0.0.1:8080/** que es la que hemos utilizado anteriormente en el entorno local simulado de Windows Azure (*Azure Development Fabric*):

```
<!-- AZURE-INTERNET-CLOUD -->
        <endpoint address="
http://nlayerazure.cloudapp.net:8080/MainModule.svc/basic"
        binding="basicHttpBinding" bindingConfiguration="BasicBindingF
orSilverlightClients"
        contract="ServiceAgent.IMainModuleService" name="BasicBindingF
orSilverlightClients" />
```

Esta definición de binding cliente debe especificarse en el fichero '***ServiceReferences.ClientConfig***' del proyecto de Silverlight (en nuestra aplicación ejemplo, el proyecto '***Silverlight.Client***').

Página por defecto del Web-Site de Silverlight en Web-Role de Azure

Si nuestra página de arranque de aplicación no se llama default.aspx, cuando intentemos entrar simplemente en la URL de la aplicación en Windows Azure, obtendremos un error, puesto que la página por defecto en Silverlight se llama '*Silverlight.Client.Web.html*'.

El portal de Windows Azure no nos permite realizar modificaciones en una consola de IIS. Pero, debido a que la mayoría de las configuraciones de IIS 7.x pueden realizarse también a través de los ficheros XML de configuración, realmente no necesitamos ninguna consola de IIS para hacer esta acción.

En este caso, podemos establecer la página por defecto de la aplicación de un WebRole de Windows Azure cambiándolo en el web.config de nuestra aplicación (**en el web.config** del **proyecto SilverlightWebRole**). Simplemente necesitamos añadir el siguiente XML de sección, dentro de la sección '**System.Web.Server**':

```
<system.webServer>
        <modules runAllManagedModulesForAllRequests="true"/>

        <defaultDocument>
          <files>
            <clear/>
            <add value="Silverlight.Client.Web.html"/>
          </files>
        </defaultDocument>

 </system.webServer>
```

Incluso, en el caso de tener varias carpetas en nuestra aplicación, también podríamos especificar una página por defecto para cada sub-carpeta. En lugar de especificarlo por "System.WebServer", lo especificamos por "location". Por ejemplo:

```
    <location path="webFolder1">
            <system.webServer>
                <defaultDocument>
                    <files>
                        <add value="myDefalutPage.aspx" />
                    </files>
                </defaultDocument>
            </system.webServer>
        </location>
        <location path="webFolder2">
            <system.webServer>
                <defaultDocument>
                    <files>
                        <add value=" myDefalutPage.aspx" />
                    </files>
                </defaultDocument>
            </system.webServer>
        </location>
```

Cadena de Conexión de *Diagnostics-Storage* de Windows Azure

En Windows Azure, para poder acceder a la información de diagnósticos (Trazas, contadores de rendimiento, etc.), deberemos exportarla de nuestras aplicaciones WebRole/WorkerRole y guardarla en un almacén de tipo 'Azure Storage'. Una vez ahí, podremos consultarlo por API o con diferentes herramientas de terceras partes (Como ***Cerebrata Azure Diagnostics Manager***).

Para ello, es importante cambiar la cadena de conexión de nuestro fichero de configuración de Role de Windows Azure. Es decir, cambiar la cadena por defecto para el entorno simulado de 'Azure Dev-Fabric', que es la siguiente línea, en el fichero '**ServiceConfiguration.cscfg**' del proyecto de configuración de Windows Azure:

```
<Setting name="DiagnosticsConnectionString" value="UseDevelopmentStorage=true" />
```

Y debemos especificar, en cambio, una cadena de conexión contra un 'Azure-Storage' que tengamos disponible para almacenar estos datos de diagnósticos, por ejemplo:

```
<ConfigurationSettings>
    <Setting name="DiagnosticsConnectionString"
    value="DefaultEndpointsProtocol=https;AccountName=cesardldiagnostics;AccountKey=hgmu0lpsPCpysC
    HANGED0OjRfR32XHGCI4pY6lOI/EvoS3yJp/9d7YJXzAAMVIzKLkyRLhKt//XNqp+CHANGED=="
    />
</ConfigurationSettings>
```

El valor de AccountName y AccountKey dependerá del cada Azure Storage de cada uno. Se puede copiar/pegar desde una página similar a la siguiente en el portal de Windows Azure:

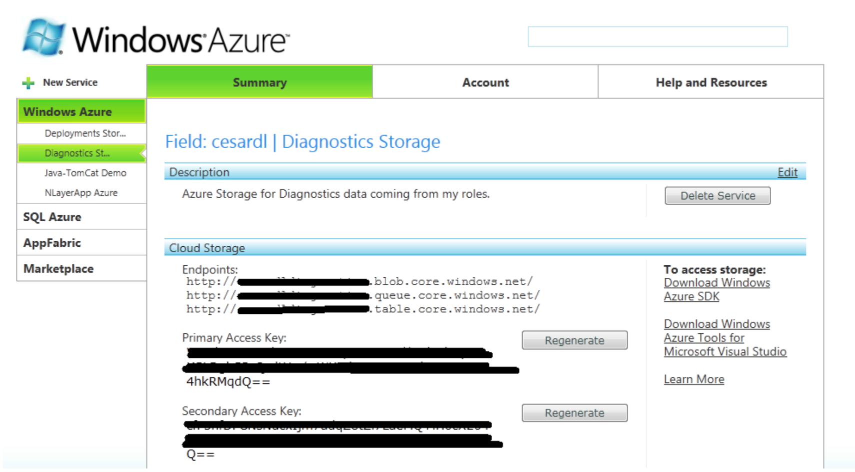


Figura 36.-

Las claves a copiar/pegar son las que ahí aparecen tachadas, lógicamente por razones de seguridad de la cuenta de Windows Azure Storage que mostramos.

También se puede cambiar mediante el interfaz de Visual Studio:

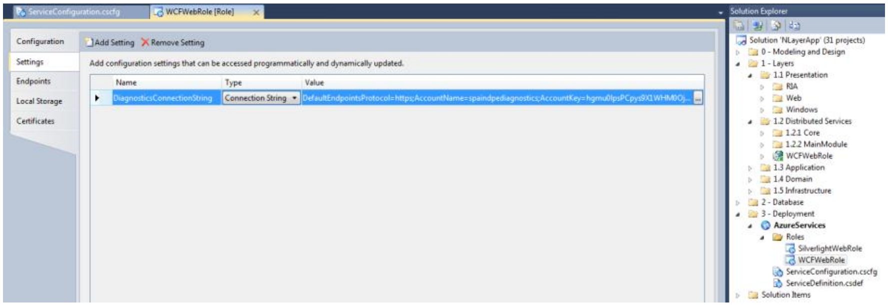


Figura 37.-

Lo que hace que se pueda recoger la información de 'Diagnostics' es precisamente el código de arranque de rol de Windows Azure, del fichero '**WebRole.cs**' de los proyectos de tipo WebRole o WorkerRole.

Es necesario tener ese código de gestión e instrumentación de Rol de Windows Azure en proyectos de Servicios WCF y proyectos Web.

Este es el único cambio un poco más intrusivo con respecto a nuestro código inicial que tenemos en Windows Server. Sin embargo es muy poco código a añadir, y como decíamos, es necesario para poder instrumentar nuestro proyecto como rol de Windows Azure (Rol Web o Rol Worker).

Precisamente cuando arrancamos el Monitor de Diagnóstico, ahí se hace uso de la URL de nuestro Azure-Storage utilizado para almacenar información de diagnósticos.

Es un código muy sencillo como el siguiente:

```
public class WebRole : RoleEntryPoint
{
        public override bool OnStart()
        {
                                        Uso de la cadena de conexión de Diagnostics

            DiagnosticMonitor.Start("DiagnosticsConnectionString");

            // For information on handling configuration changes
            // see the MSDN topic at http://go.microsoft.com/fwlink/?LinkId=
166357.
            RoleEnvironment.Changing += RoleEnvironmentChanging;

            return base.OnStart();
        }

        private void RoleEnvironmentChanging(object sender, RoleEnvironmentC
hangingEventArgs e)
        {
            // If a configuration setting is changing
            if (e.Changes.Any(change => change is RoleEnvironmentConfigurati
onSettingChange))
```

```
            {
                // Set e.Cancel to true to restart this role instance
                e.Cancel = true;
            }
        }
}
```

Para conocer mejor el sistema de instrumentalización de Windows Azure, la siguiente información:

Take Control of Logging and Tracing in Windows Azure
http://msdn.microsoft.com/en-us/magazine/ff714589.aspx

3.5.5.- Gestión de imágenes en Web: Cambio de almacén local (disco) a Windows Azure Blobs

Especialmente para aplicaciones Web que hacen uso de imágenes (.JPG, .PNG, etc.) la mayoría de las veces almacenadas en disco duro local, es muy recomendable migrar ese sistema a un sistema de almacenamiento centralizado de dichas imágenes en ***Blobs*** de *Windows Azure Storage*. Esto sobre todo es muy importante si las imágenes son dinámicas, cambiantes, etc., como por ejemplo imágenes de fotos de un catálogo de productos. Si se trata de imágenes estáticas que no cambian nunca, pueden permanecer en el directorio local junto a los ficheros de recursos de ASP.NET.

Para más información sobre 'Windows Azure BlobStorage', consultar el SDK de Windows Azure en:

http://www.microsoft.com/downloads/details.aspx?FamilyID=21910585-8693-4185-826e-e658535940aa&displaylang=en

3.5.6.- Seguridad en Windows Azure

Este punto será probablemente el más complejo de todos los anteriores, pues al tener nuestra aplicación en Internet fuera del dominio de seguridad de nuestra organización, probablemente el sistema de seguridad de la aplicación tenga que cambiar por lo menos parcialmente.

Dependiendo del tipo de seguridad (autenticación y autorización) que estemos usando anteriormente en nuestra aplicación *on-premise*, deberemos realizar algunos cambios parciales o a veces incluso ningún cambio:

Seguridad App-On-Premise	Seguridad App en Windows Azure
Membership como sistema de autenticación contra providers aislados (Tablas de B.D. SQL Server, propias de la aplicación)	Uso también de **Membership**. No requiere cambios que impacten en el desarrollo de la aplicación, puesto que el provider de Membership para SQL Azure es el mismo que para SQL Server. Solamente se deberán crear las tablas requeridas en SQL Azure. Para ello el Script SQL es un poco diferente. Ver: **http://code.msdn.microsoft.com/KB2006191** **http://support.microsoft.com/kb/2006191/**
Seguridad Windows integrada con autenticación de **Active Directory** y API de autorización del framework o de AzMan (Authorization Manager).	Cambio a '**Seguridad con orientación a *claims***' con **WIF** (*Windows Identity Foundation*) e integración con el AD corporativo mediante ADFS 2.0 como STS corporativo publicado en Internet. Aun en el caso de que los usuarios no sean corporativos, actualmente no es posible crear un AD aislado en Windows Azure, por eso se debe delegar la autenticación en un AD corporativo a través de ADFS 2.0 como STS (*Security Token Service*). *NOTA*: Para realizar cambios relacionados con WIF en nuestra aplicación y para establecer relaciones de confianza entre nuestra aplicación y el STS (ADFS 2.0), es muy conveniente hacer uso de la utilidad **FedUtil**, proporcionada como parte de WIF http://msdn.microsoft.com/en-us/library/ee517284.aspx
Sistemas de autenticación publicados/accesibles desde Internet, bien **Windows Live ID**, **OpenID**, o incluso **Seguridad Windows** integrada con autenticación de **Active Directory** publicado con ADFS 2.0 en Internet.	Uso de Windows Azure ***AppFabric Access Control.*** Especialmente en los casos en los que deseemos disponer de varios Sistemas de autenticación compatibles e incluso simultáneos para nuestra aplicación, aquí es cuando más nos interesa hacer uso de AD, pues nos desacopla nuestra aplicación de los diferentes sistemas de autenticación que queramos usar. Para más información, revisar los apartados de esta guía relacionados con *Seguridad Orientada a Claims*, muy relacionado con AC (AC es en definitiva un STS en la nube).

3.5.7.- Otros puntos a tener en cuenta al migrar aplicaciones a Windows Azure

Lo siguiente son algunos consejos de migración, simples pero útiles:

- **Longitud de nombres de ensamblados**: Tener cuidado con la longitud de los nombres de nuestros ensamblados. Nombres largos que en una solución para Windows Server no dan ningún problema, en Windows Azure pueden

ocasionar problemas por sobrepasarse el límite máximo al añadirle más información Windows Azure.

- **Uso de Azure Diagnostics Monitor TraceListener**: Si se está haciendo uso de *listeners* para realizar trazas y *logging*, debemos cambiarlo para que se haga uso de Microsoft.WindowsAzure.Diagnostics.DiagnosticMonitorTraceListener como listener por defecto. Es recomendable hacer uso de un 'sharedListener' que apunte al de Windows Azure. Por encima podremos seguir haciendo uso de nuestro sistema anterior.

- **Sesiones ASP.NET**: No debemos hacer uso de sesiones ASP.NET '*in process*' en el espacio de memoria de los pool de IIS, pues cuando tengamos varios roles balanceados, el sistema de sesiones funcionará mal. Esto ocurre igual en sistemas balanceados en servidores Windows Server si el balanceo es puro (Sin afinidad).

    - Opciones actuales en Windows Azure:

        - *ASP.NET Session provider for Azure Storage (Sample)* : **http://code.msdn.microsoft.com/windowsazuresamples**

    - Opciones futuras en Windows Azure:

        - *ASP.NET Session provider for SQL Azure* (Actualmente este provider no funciona en SQL Azure porque hace uso de SQL Agent el cual no está soportado en SQL Azure)

        - *ASP.NET Session provider for AppFabric-Cache* (Actualmente este *provider* está disponible solo para *Windows Server*, puesto que *AppFabric-Cache* (*codename.'Velocity'*) solo está disponible por ahora para *Windows Server*). Pero será el más recomendado, en breve.

- **Reintentos en Conexiones de SQL Azure**: En SQL Azure, para conseguir alta disponibilidad, muchas veces se cierran conexiones, por lo que la lógica de nuestra aplicación tiene que controlar errores de conexión y realizar intentos de re-conexión. Ver:
  http://blogs.msdn.com/b/sqlazure/archive/2010/05/11/10011247.aspx

4.- ESCENARIO AVANZADO: APLICACIÓN ESCALABLE EN CLOUD-COMPUTING

Este escenario nos va a requerir realizar cambios en nuestra arquitectura lógica, definiendo nuevos patrones de arquitectura y diseño indicados para aplicaciones con altos requerimientos de escalabilidad (como el patrón *CQRS – Command and Query*

Responsability Segregation y otros patrones) e implementándolos en algunas ocasiones con tecnologías nativas/exclusivas de nuestra plataforma de Cloud-Computing.

Los requerimientos de alta escalabilidad también pueden impactar en las tecnologías de persistencia y acceso a datos e incluso en las propias fuentes de datos.

4.1.- Arquitectura Lógica (Escenario Avanzado en la nube)

En el caso de hacer uso de la nube por requerimientos de una alta escalabilidad, la arquitectura lógica de nuestra aplicación se verá probablemente impactada, pues será muy recomendable hacer uso de ciertos patrones de Arquitectura y diseño que nos favorecerán dicha alta escalabilidad.

Los patrones y arquitecturas que recomendamos evaluar e implementar para estos escenarios avanzados, son:

- *CQRS – Command and Query Responsability Segregation*
- *Event Sourcing y Event Stores*
- *EDA – Event Driven Architecture*

Antes de entrar en detalle, es importante destacar que los siguientes patrones no son exclusivos para aplicaciones en 'Cloud-Computing'. De hecho su definición original ha sido identificada y definida de forma independiente e incluso anterior al fenómeno de *cloud-computing*. Sin embargo, son patrones de arquitectura y diseño que encajan muy bien con los objetivos de la nube, especialmente con requerimientos de alta escalabilidad. Pero podrían ser perfectamente implementados y utilizados en aplicaciones '*On-Premise*'.

Como en otros apartados anteriores, exponemos inicialmente una explicación lógica independiente de la tecnología y después pasaremos a mostrar una posible implementación con tecnologías Microsoft.

4.2.- Patrón CQRS (*Command and Query Responsibility Segregation*)

El patrón **CQRS** (*Segregación de Responsabilidades de Consultas y Comandos*) realmente viene a incidir y estructurar la idea explicada anteriormente (Cap.3: 'Acceso Dual a Datos').

CQRS es una forma de diseñar arquitecturas de aplicaciones empresariales que requieren:

- Escalabilidad masiva.
- Foco en el negocio y no en la tecnología.
- Poder crecer y gestionar problemas complejos sin crecer exponencialmente en costes de desarrollo.
- Adaptarse a requerimientos de negocio cambiantes
- Beneficiarse de los recursos típicos de cloud-Computing

Uno de los mayores exponentes de este patrón es, actualmente, ***Greg Young***, el cual establece la siguiente frase, completamente lógica:

"*Un único modelo no puede ser apropiado para realizar informes, búsquedas y comportamientos transaccionales*".

El principal concepto propuesto por CQRS es el concepto de separación entre las tareas de consultas y las tareas transaccionales/persistencias (edición/inserción/borrado, etc.).

El siguiente esquema muestra de forma simplificada el patrón CQRS:

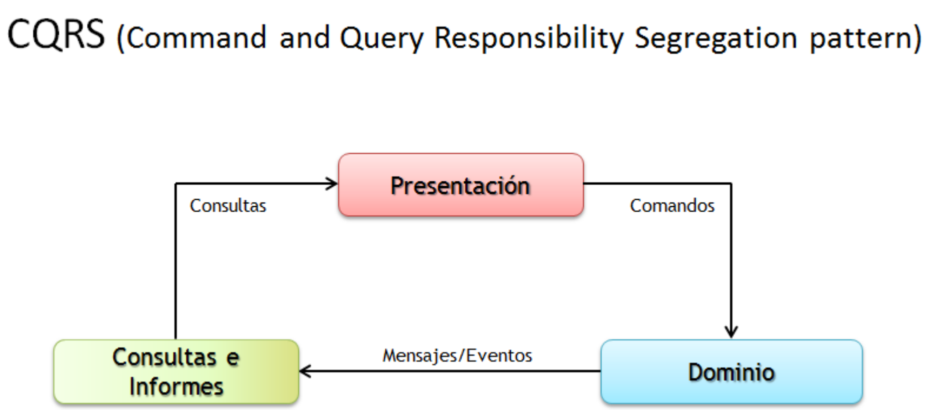


Figura 38.-

En este funcionamiento, CQRS propone el uso de *comandos*, implementados como **mensajes** o **eventos** como mecanismo de disparo y realización de acciones de tipo 'actualización de datos', normalmente relacionados con lógica de negocio y transacciones. Pero separando las acciones del Dominio y persistencia en un almacén específico para las actualizaciones (normalmente almacén transaccional) de otra área y almacén de datos dedicado solo para las consultas/lecturas.

CQRS es una aproximación que proporciona soluciones a los siguientes problemas típicos en el desarrollo de aplicaciones:

- Escalabilidad y cuellos de botella en el rendimiento.
- Conflictos de concurrencia, resolución y prevención.
- Datos estancados ('*staleness*').
- Complejidad de diseño, desarrollo y mantenimiento.

El resumen de los **puntos fundamentales de CQRS** es:

- Todos los cambios de estado se articulan con Mensajes/Eventos.
- Los Servicios de Aplicación reciben **Comandos** y publican **Mensajes/Eventos**.
- Las Consultas (Listados e Informes) se actualizan como resultado de la publicación de los Mensajes/Eventos.
- Todas las Consultas de la Capa de Presentación van directamente contra el sistema de *Consultas/Reporting*. El Dominio no está aquí involucrado en absoluto.

Y un ejemplo concreto sería:

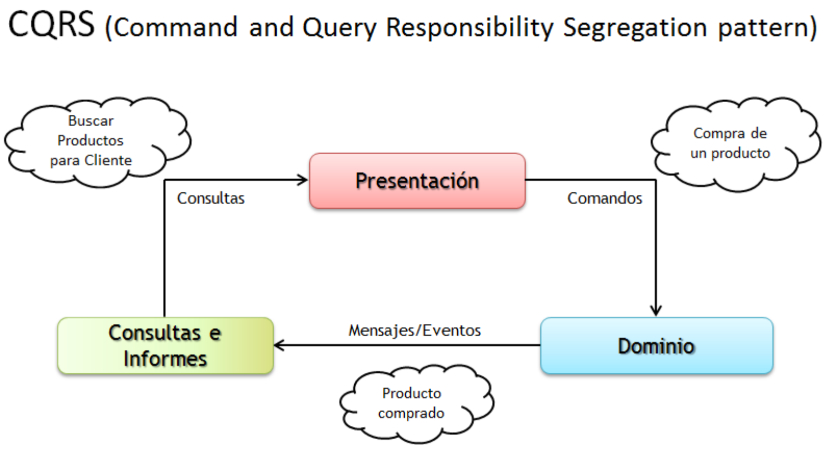


Figura 39.- CQRS (Command and Query Responsibility Segregation Pattern)

Sobre la implementación de Comandos, en el caso de hacer uso de eventos, estaríamos haciendo uso de '*Event-Sourcing*' e podríamos derivar también hacia patrones de **EDA** (*Event Driven Architecture*). Sin embargo, realmente el patrón CQRS es independiente de la implementación de los *commands*, es decir, podemos implementar *commands* como queramos, pueden ser eventos o pueden ser, y en muchos casos es más útil, mensajes basados en colas de mensajes, o ambas cosas a la vez.

Y un funcionamiento ya más detallado de CQRS sería algo similar al siguiente esquema:

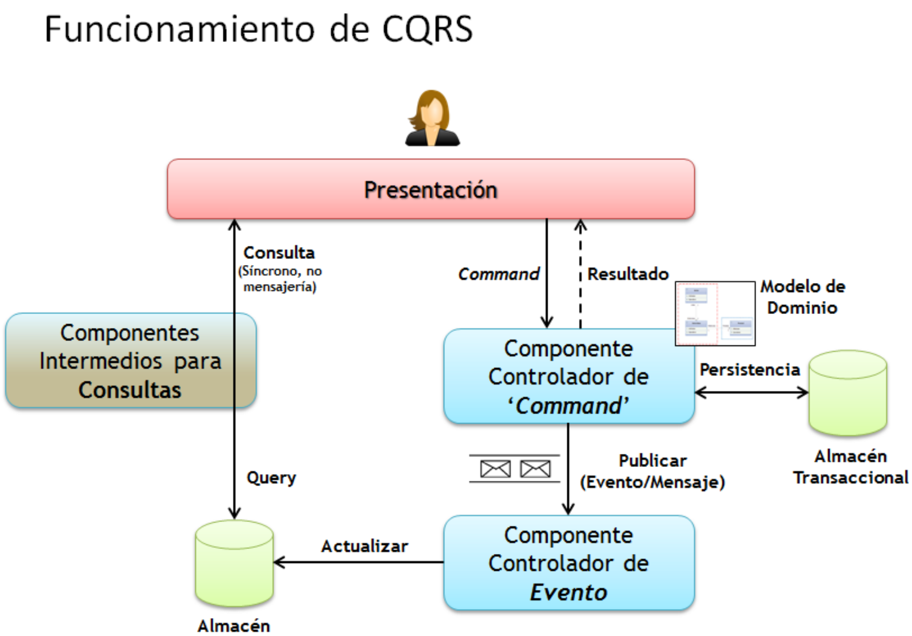


Figura 40.- Funcionamiento de CQRS

Por supuesto, en el esquema anterior se están obviando mecanismos de invocación remota y otros componentes intermedios que puedan hacer falta. Solo mostramos los componentes fundamentales de la idea.

Así mismo, reiterar que el cómo se procesan los *Commands*, son detalles del diseño final y de la implementación. Pueden implementarse con eventos o con mensajes (colas de mensajes tipo *MSMQ* o *Azure-Messages*) e incluso con ambos (Evento+Mensaje).

En los escenarios donde encaja muy bien **CQRS** es en **escenarios de alta escalabilidad** muy afines a '*Cloud-Computing*', donde un desacoplamiento entre el almacén transaccional y el almacén de consultas puede ser muy ventajoso y donde incluso la naturaleza del almacén transaccional puede ser muy diferente a la naturaleza del almacén dedicado solo a las consultas.

4.2.1.- ¿Por qué CQRS?

Uno de los mejores valores del patrón CQRS es que consigue solucionar bastante bien los problemas de incompatibilidad entre los siguientes puntos de arquitectura, los cuales normalmente se interfieren entre ellos:

- Consistencia
- Disponibilidad
- Particionabilidad

La realidad es que normalmente en una aplicación solo se puede garantizar dos de los tres puntos anteriores. Si se añade el tercero, normalmente tenemos que quitar alguno de los otros dos, porque se ven afectados o impactados. Por ejemplo, podemos tener un sistema con alta disponibilidad y consistente (basado en un SGBD relacional como SQL Server y una arquitectura típica N-Layer), pero su particionabilidad para poder ser más escalable probablemente sea baja. O por el contrario, podemos tener un sistema con alta disponibilidad y particionable pero entonces tendremos que ceder en el parámetro de la consistencia.

Con CQRS, al dividir el sistema en varias piezas (Consultas y Comandos) podemos ajustar estos parámetros anteriores de forma separada e independiente. Por supuesto que las incompatibilidades anteriormente expuestas seguirán existiendo, pero al tenerlas localizadas en diferentes áreas, podemos tomar decisiones y optimizar nuestro sistema en diferentes áreas de una forma más independiente.

En definitiva, al tener el sistema dividido en dos áreas muy independientes (incluso con diferente fuente de datos cada una), una con los comandos (operaciones de actualizaciones y transacciones) y otra con las consultas, podremos tomar decisiones por separado. Podemos por ejemplo tener consistencia y transacciones ACID en las actualizaciones y en cambio podemos tener particionabilidad en el área de consultas y conseguir ahí un nivel muy diferente de escalabilidad, que es donde realmente está normalmente el gran volumen de operaciones de los usuarios, en las consultas/lecturas.

Y desde el punto de vista del área de lecturas/consultas. Si ya tenemos consistencia completa en el área de actualizaciones (comandos en CQRS), ¿realmente necesitamos en nuestro sistema también consistencia completa en el área de lecturas/consultas? ¿Si el modelo de lecturas tuviera sus datos actualizados unos 5 segundos por detrás del modelo de actualizaciones/comandos, eso sería aceptable? En la mayoría de los casos, sí sería aceptable. Y probablemente se pueda conseguir que ni siquiera sean 5 segundos, sean 2 o 1 segundo. Esos tiempos son muy pequeños desde el punto de vista del uso de la aplicación por parte de los usuarios y en la mayoría de los escenarios, es aceptable. Esta cesión, sin embargo, podría permitir a nuestro modelo de consultas/lecturas ser altamente particionable y por lo tanto con una escalabilidad mucho mayor que si estuviera acoplado al sistema de actualizaciones. Muy poco coste desde el punto de vista de usuario comparado con los beneficios conseguidos a nivel de la escalabilidad.

Lo importante en esta discusión es volver a recalcar que con CQRS podremos tomar/cambiar decisiones de forma independiente en el sistema de actualizaciones/transaccional con respecto al sistema de consultas/lecturas (o viceversa), lo que nos dará mucho más juego para poder optimizar nuestro sistema de la mejor forma posible. Esas decisiones independientes pueden ser desde el tipo de almacén o fuente de datos (por ejemplo un SGBD relacional como *SQL Server* o *SQL Azure* para el área transaccional/comandos y una fuente de datos no relacional como *Azure-Storage* o incluso un sistema *OLAP* como almacén para los datos del área de lectura/consultas). Incluso podríamos llegar a separar físicamente el sistema de lecturas del sistema de actualizaciones (uno en la nube y otro en *on-premise*, etc.). Esta libertad de optimización es fundamental, porque cada implantación de aplicación puede tener

unos requerimientos radicalmente diferentes dependiendo de las necesidades de los usuarios.

Por último, resumimos los **beneficios fundamentales de CQRS**:

- ‘Dominio’ completamente encapsulado y que solo expone comportamientos
- Las consultas no hacen uso del ‘Modelo de Dominio’
- No hay desajuste de impedancias entre objetos y datos relacionales
- Sistema perfecto para Trazabilidad de históricos y auditorías
- Fácil integración con sistemas externos
- Rendimiento y especialmente **una gran escalabilidad**

Conclusiones

En estas páginas hemos intentado plasmar los principales retos a los que se enfrentan las empresas en el desarrollo de aplicaciones empresariales complejas y de larga duración. Aunque el libro refleja los principales problemas de este tipo de desarrollos y ofrece soluciones a los mismos, hay muchos aspectos en los que se podría haber profundizado más y que darían para otro libro del mismo tamaño. Por esta razón aconsejamos al lector utilizar este libro como punto de referencia para la construcción de aplicaciones empresariales y le animamos a extender el contenido de este libro con otros textos de referencia como '*Domain-Driven Design Tackling Complexity in the heart of Software*' de Eric Evans.

Estamos convencidos de que el desarrollo de buen software pasa por el conocimiento profundo del dominio del sistema en construcción. Alentamos al lector a que cambie su forma de trabajo y se centre en construir buenos modelos de dominio, y a que use las tecnologías desarrolladas por Microsoft para facilitar la construcción del sistema. El valor del software está en el dominio que modela, y que permite atender las necesidades de unos determinados clientes, la tecnología existe para facilitar el desarrollo de las aplicaciones en base a los modelos de dominio que construimos, y no para dificultar su desarrollo, y este es un aspecto que hemos querido dejar claro.

También queremos decir que el futuro de IT se encuentra en la nube, y en ella encontramos nuevos problemas a los que hacer frente. Animamos al lector a investigar sobre '*Event Sourcing'*, '*Command and Query Responsibility Segregation'*, y a explorar todo este conjunto de patrones ideales para escenarios de alta escalabilidad que encajan perfectamente con '*Cloud Computing*' y el *PaaS* de Microsoft, es decir, *Windows Azure*.

Por último, queremos dar las gracias a todos los lectores que han seguido el desarrollo de este proyecto (que seguiremos evolucionando, es algo vivo, ¡no ha acabado!) así como a los colaboradores y desarrolladores que han hecho uso de nuestra aplicación ejemplo en CODEPLEX (*NLayerApp*) siguiéndolo como modelo y referencia para sus propias aplicaciones/productos. Toda vuestra contribución de preguntas y *feedback* sin duda ha hecho muchas veces replantear cosas y en definitiva, mejorar la calidad de la propuesta. Sabemos que no se puede estar de acuerdo completamente en un tema, y menos en arquitectura software, pero esperamos que estas páginas hayan enriquecido la perspectiva del diseño y desarrollo de aplicaciones de una parte importante de las comunidades de arquitectos de software y desarrolladores.

Atentamente,

'El equipo A'. ☺

(A de Arquitectura, ¿qué pensabas?)